



# Preliminary in-progress RISC-V "P" Extension

Version 0.01.01-draft-20230915

This document is in the Development state. Assume anything can change.

# Table of Contents

## Contents

Chapter 1. Introduction .....	1
Chapter 2. Shorthand Definitions and .....	2
2.1. Shorthand Definitions.....	2
2.2. Terminology .....	3
Chapter 3. RISC-V P Extension Instruction.....	4
3.1. SIMD Data Processing Instructions.....	4
3.2. Partial-SIMD Data Processing Instructions .....	32
3.3. 64-bit Data Computation Instructions.....	47
3.4. Non-SIMD Instructions.....	58
3.5. RV64 Only Instructions.....	67
Chapter 4. Instructions Duplicated with.....	81
Chapter 5. P Extension Subsets (hidden for.....	82
Chapter 6. Detailed Instruction Descriptions.....	83
Chapter 7. Detailed Instruction Descriptions.....	84
7.1. PADD.B (ADD8) (SIMD 8-bit Addition).....	85
7.2. PADD.H (ADD16) (SIMD 16-bit Addition) .....	86
7.3. PMSEQ.B (CMPEQ8) (SIMD 8-bit Integer Compare.....	87
7.4. PMSEQ.H (CMPEQ16) (SIMD 16-bit Integer Compare.....	88
7.5. PAS.HX (CRAS16) (SIMD 16-bit Cross.....	89
7.6. PSA.HX (CRSA16) (SIMD 16-bit Cross.....	90
7.7. PABS.H (KABS16) (SIMD 16-bit Unsigned Absolute) .....	91
7.8. PSADD.B (KADD8) (SIMD 8-bit Signed Saturating.....	92
7.9. PSADD.H (KADD16) (SIMD 16-bit Signed.....	93
7.10. SADD (KADDW) (Signed Addition with Q31 .....	94
7.11. PSAS.HX (KCRAS16) (SIMD 16-bit Signed .....	95
7.12. PSSA.HX (KCRSA16) (SIMD 16-bit Signed .....	97
7.13. PMULQ.H (KHM16), KHMX16 .....	99
7.14. PMACC.W.HEE (KMABB), PMAC.W.HEO (KMABT), .....	101
7.15. MHACC/PMHACC.W (KMMAC), MHRACC/PMHRACC.W (KMMAC.u) .....	103
7.16. PMHACC.W.HE (KMMAWB), KMMAWB.u.....	105
7.17. PMHACC.W.HO(KMMAWT), KMMAWT.u .....	107
7.18. PSSUB.B (KSUB8) (SIMD 8-bit Signed Saturating .....	109
7.19. SSUB (KSUBW) (Signed Subtraction with Q31.....	110
7.20. PSSUB.H (KSUB16) (SIMD 16-bit Signed.....	111
7.21. MULQ/PMULQ.W (KWMMUL), MULQR/PMULQR.W (KWMMUL.u) .....	112
7.22. Need new opcodes (MULR64).....	114
7.23. Alias for MUL.WEE - MULSR64 (Multiply Word.....	116
7.24. PDIFSUMU.B (PBSAD) (Parallel Byte Sum of.....	118
7.25. PDIFSUMAU.B (PBSADA) (Parallel Byte.....	119

7.26. PAADD.B (RADD8) (SIMD 8-bit Signed Averaging .....	120
7.27. PAADD.H (RADD16) (SIMD 16-bit Signed Averaging.....	121
7.28. AADD (RADDW) (32-bit Signed Averaging.....	122
7.29. PAAS.HX (RCRAS16) (SIMD 16-bit Signed.....	123
7.30. PASA.HX (RCRSA16) (SIMD 16-bit Signed .....	124
7.31. PASUB.B (RSUB8) (SIMD 8-bit Signed Averaging.....	125
7.32. PASUB.H (RSUB16) (SIMD 16-bit Signed Averaging.....	126
7.33. ASUB (RSUBW) (32-bit Signed Averaging.....	127
7.34. PMSGE.B (SCMPEL8) (SIMD 8-bit Signed.....	128
7.35. PMSGE.H (SCMPEL16) (SIMD 16-bit Signed .....	129
7.36. PMSLT.B (SCMLT8) (SIMD 8-bit Signed Compare.....	130
7.37. PMSLT.H (SCMLT16) (SIMD 16-bit Signed .....	131
7.38. PM4ADDA.H (SMALDA), SMALXDA.....	132
7.39. PM2ADDA.W (SMAR64) (Signed Multiply and.....	134
7.40. PMAX.B (SMAX8) (SIMD 8-bit Signed Maximum).....	136
7.41. PMAX.H (SMAX16) (SIMD 16-bit Signed Maximum).....	137
7.42. PMUL.W.HEE (SMBB16), PMUL.W.HEO (SMBT16), .....	138
7.43. PMIN.B (SMIN8) (SIMD 8-bit Signed Minimum).....	140
7.44. PMIN.H (SMIN16) (SIMD 16-bit Signed Minimum).....	141
7.45. PMULH.W (SMMUL), MULHR/PMULHR.W (SMMUL.u) .....	142
7.46. PMULH.W.HE (SMMWB), SMMWB.u .....	144
7.47. PMULH.W.HO (SMMWT), <a href="#">SMMWT.u</a> .....	146
7.48. PSUB.B (SUB8) (SIMD 8-bit Subtraction).....	148
7.49. PSUB.H (SUB16) (SIMD 16-bit Subtraction).....	149
7.50. SUBD (SUB64) (64-bit Subtraction).....	150
7.51. PMSGEU.B (UCMPEL8) (SIMD 8-bit Unsigned).....	151
7.52. PMSGEU.H (UCMPEL16) (SIMD 16-bit Unsigned) .....	152
7.53. PMSLTU.B (UCMLT8) (SIMD 8-bit Unsigned).....	153
7.54. PMSLTU.H (UCMLT16) (SIMD 16-bit Unsigned).....	154
7.55. PSADDU.B (UKADD8) (SIMD 8-bit.....	155
7.56. PSADDU.H (UKADD16) (SIMD 16-bit .....	156
7.57. SADDU (UKADDW) (Unsigned .....	157
7.58. PSSUBU.B (UKSUB8) (SIMD 8-bit.....	158
7.59. PSSUBU.H (UKSUB16) (SIMD 16-bit .....	159
7.60. SSUBU (UKSUBW) (Unsigned Subtraction.....	160
7.61. PMAXU.B (UMAX8) (SIMD 8-bit Unsigned).....	161
7.62. PMAXU.H (UMAX16) (SIMD 16-bit Unsigned) .....	162
7.63. PSMINU.B (UMIN8) (SIMD 8-bit Unsigned).....	163
7.64. PMINU.H (UMIN16) (SIMD 16-bit Unsigned) .....	164
7.65. PAADDU.B (URADD8) (SIMD 8-bit Unsigned).....	165
7.66. PAADDU.H (URADD16) (SIMD 16-bit.....	166
7.67. AADDU (URADDW) (32-bit Unsigned Averaging .....	167

7.68. PASUBU.B (URSUB8) (SIMD 8-bit.....	168
7.69. PASUBU.H (URSUB16) (SIMD 16-bit.....	169
7.70. ASUBU (URSUBW) (32-bit Unsigned Averaging .....	170
Chapter 8. Detailed Instruction Descriptions.....	171
8.1. PADD.W (ADD32) (SIMD 32-bit Addition).....	172
8.2. PAS.WX (CRAS32) (SIMD 32-bit Cross Addition &.....	173
8.3. PSA.WX (CRSA32) (SIMD 32-bit Cross Subtraction &.....	174
8.4. PABS.W (KABS32) (Scalar 32-bit Absolute.....	175
8.5. PSADD.W (KADD32) (SIMD 32-bit Signed Saturating.....	176
8.6. PSAS.WX (KCRAS32) (SIMD 32-bit Signed.....	177
8.7. PSSA.WX (KCRSA32) (SIMD 32-bit Signed.....	178
8.8. MACC.WEE (KMABB32), MACC.WEO (KMABT32),.....	179
8.9. <b>KMADA32</b> , PM2ADDA.WX (KMAXDA32).....	181
8.10. PM2ADD.W (KMDA32), PM2ADD.WX (KMXDA32).....	183
8.11. <b>KMADS32</b> , PM2SUBA.W (KMADRS32), PM2SUBA.WX .....	185
8.12. PSSUB.W (KSUB32) (SIMD 32-bit Signed.....	187
8.13. PAADD.W (RADD32) (SIMD 32-bit Signed Averaging.....	188
8.14. PAAS.WX (RCRAS32) (SIMD 32-bit Signed.....	189
8.15. PASA.WX (RCRSA32) (SIMD 32-bit Signed .....	190
8.16. PASUB.W (RSUB32) (SIMD 32-bit Signed Averaging.....	191
8.17. PMAX.W (SMAX32) (SIMD 32-bit Signed Maximum).....	192
8.18. MUL.WEE (SMBB32), MUL.WEO (SMBT32),.....	193
8.19. <b>SMDS32</b> , PM2SUB.W (SMDRS32), PM2SUB.WX .....	195
8.20. PMIN.W (SMIN32) (SIMD 32-bit Signed Minimum).....	197
8.21. PSUB.W (SUB32) (SIMD 32-bit Subtraction).....	198
8.22. PSADDU.W (UKADD32) (SIMD 32-bit .....	199
8.23. PSSUBU.W (UKSUB32) (SIMD 32-bit.....	200
8.24. PMAXU.W (UMAX32) (SIMD 32-bit Unsigned .....	201
8.25. PMINU.W (UMIN32) (SIMD 32-bit Unsigned .....	202
8.26. PAADDU.W (URADD32) (SIMD 32-bit.....	203
8.27. PASUBU.W (URSUB32) (SIMD 32-bit.....	204
Chapter 9. New User Control & Status.....	205
9.1. Fixed-point Saturation Flag Register.....	206
Chapter 10. Instruction Encoding Table .....	207
Chapter 11. Removed Instructions Due to .....	213
Appendix A: Instruction Latency and.....	214
Appendix B: instructions from original draft .....	215
1. <b>ADDD</b> (ADD64) (64-bit Addition).....	215
2. <b>AVE</b> (Average with Rounding).....	216
3. <b>BITREV</b> (Bit Reverse).....	217
4. <b>BITREVI</b> (Bit Reverse Immediate).....	218
5. <b>CLROV</b> (Clear OV flag).....	220

6.	CLRS8 (SIMD 8-bit Count Leading Redundant Sign) .....	221
7.	CLRS16 (SIMD 16-bit Count Leading Redundant Sign) .....	223
8.	CLRS32 (SIMD 32-bit Count Leading Redundant Sign) .....	225
9.	CLZ8 (SIMD 8-bit Count Leading Zero) .....	227
10.	CLZ16 (SIMD 16-bit Count Leading Zero) .....	229
11.	CLZ32 (SIMD 32-bit Count Leading Zero) .....	231
12.	INSB (Insert Byte) .....	233
13.	KABS8 (SIMD 8-bit Saturating Absolute) .....	234
14.	KADD64 (64-bit Signed Saturating Addition) .....	236
15.	KADDH (Signed Addition with Q15 Saturation) .....	238
16.	KDMBB, KDMBT, KDMTT .....	238
17.	KDMABB, KDMABT, KDMATT .....	242
18.	KHM8, KHMX8 .....	245
19.	KHMBB, KHMBT, KHM TT .....	248
20.	KMADA, KMAXDA .....	251
21.	KMADS, KMADRS, KMAXDS .....	254
22.	KMAR64 (Signed Multiply and Saturating Add to) .....	257
23.	KMDA, KMXDA .....	259
24.	KMMAWB2, KMMAWB2.u .....	262
25.	KMMAWT2, KMMAWT2.u .....	265
26.	KMMSB, KMMSB.u .....	268
27.	KMMWB2, KMMWB2.u .....	271
28.	KMMWT2, KMMWT2.u .....	274
29.	KMSDA, KMSXDA .....	277
30.	KMSR64 (Signed Multiply and Saturating Subtract) .....	280
31.	KSLLW (Saturating Shift Left Logical for Word) .....	282
32.	KSLLIW (Saturating Shift Left Logical Immediate) .....	283
33.	KSLL8 (SIMD 8-bit Saturating Shift Left Logical) .....	284
34.	KSLLI8 (SIMD 8-bit Saturating Shift Left Logical) .....	286
35.	KSLL16 (SIMD 16-bit Saturating Shift Left) .....	288
36.	KSLLI16 (SIMD 16-bit Saturating Shift Left Logical) .....	290
37.	KSLRA8, KSLRA8.u .....	292
38.	KSLRA16, KSLRA16.u .....	295
39.	KSLRAW (Shift Left Logical with Q31 Saturation) .....	298
40.	KSLRAW.u (Shift Left Logical with Q31 Saturation) .....	300
41.	KSTAS16 (SIMD 16-bit Signed Saturating Straight) .....	302
42.	KSTSA16 (SIMD 16-bit Signed Saturating Straight) .....	304
43.	KSUB64 (64-bit Signed Saturating Subtraction) .....	306
44.	KSUBH (Signed Subtraction with Q15 Saturation) .....	308
45.	MADDR32 (Multiply and Add to 32-Bit Word) .....	309
46.	MSUBR32 (Multiply and Subtract from 32-Bit) .....	310
47.	PKBB16, PKBT16, PKTT16, PKTB16 .....	311

48.	RDOV (Read OV flag) .....	314
49.	RSTAS16 (SIMD 16-bit Signed Averaging Straight) .....	315
50.	RSTA16 (SIMD 16-bit Signed Averaging Straight) .....	317
51.	RSUB64 (64-bit Signed Averaging Subtraction) .....	319
52.	ABS (KABSW) (Scalar 32-bit Absolute Value) .....	321
53.	RADD64 (64-bit Signed Averaging Addition) .....	323
54.	SCLIP8 (SIMD 8-bit Signed Clip Value) .....	325
55.	SCLIP16 (SIMD 16-bit Signed Clip Value) .....	327
56.	SCLIP32 (SIMD 32-bit Signed Clip Value) .....	329
57.	SLL8 (SIMD 8-bit Shift Left Logical) .....	331
58.	SLLI8 (SIMD 8-bit Shift Left Logical Immediate) .....	333
59.	SLL16 (SIMD 16-bit Shift Left Logical) .....	334
60.	SLLI16 (SIMD 16-bit Shift Left Logical Immediate) .....	336
61.	SMAL (Signed Multiply Halfs & Add 64-bit) .....	337
62.	SMALBB, SMALBT, SMALTT .....	339
63.	SMALDS, SMALDRS, SMALXDS .....	343
64.	SMAQA (Signed Multiply Four Bytes with 32-bit Adds) .....	347
65.	SMAQA.SU (Signed and Unsigned Multiply Four) .....	349
66.	SMDS, SMDRS, SMXDS .....	351
67.	SMSLDA, SMSLXDA .....	354
68.	SMSR64 (Signed Multiply and Subtract from 64-) .....	357
69.	SMUL8, SMULX8 .....	359
70.	SMUL16, SMULX16 .....	363
71.	SRA.u (Rounding Shift Right Arithmetic) .....	366
72.	SRAI.u (Rounding Shift Right Arithmetic) .....	368
73.	SRA8, SRA8.u .....	370
74.	SRAI8, SRAI8.u .....	372
75.	SRA16, SRA16.u .....	374
76.	SRAI16, SRAI16.u .....	376
77.	SRL8, SRL8.u .....	378
78.	SRLI8, SRLI8.u .....	380
79.	SRL16, SRL16.u .....	382
80.	SRLI16, SRLI16.u .....	384
81.	STAS16 (SIMD 16-bit Straight Addition &) .....	386
82.	STSA16 (SIMD 16-bit Straight Subtraction &) .....	388
83.	SUNPKD810, SUNPKD820, SUNPKD830, .....	390
84.	SWAP8 (Swap Byte within Halfword) .....	393
85.	SWAP16 (Swap Halfword within Word) .....	394
86.	UCLIP8 (SIMD 8-bit Unsigned Clip Value) .....	395
87.	UCLIP16 (SIMD 16-bit Unsigned Clip Value) .....	396
88.	UCLIP32 (SIMD 32-bit Unsigned Clip Value) .....	397
89.	UKADD64 (64-bit Unsigned Saturating Addition) .....	399

90.	UKADDH (Unsigned Addition with U16)	401
91.	UKCRAS16 (SIMD 16-bit Unsigned Saturating)	402
92.	UKRSA16 (SIMD 16-bit Unsigned Saturating)	404
93.	UKMAR64 (Unsigned Multiply and Saturating)	406
94.	UKMSR64 (Unsigned Multiply and Saturating)	408
95.	UKSTAS16 (SIMD 16-bit Unsigned Saturating)	410
96.	UKSTA16 (SIMD 16-bit Unsigned Saturating)	412
97.	UKSUB64 (64-bit Unsigned Saturating Subtraction)	414
98.	UKSUBH (Unsigned Subtraction with U16)	416
99.	UMAR64 (Unsigned Multiply and Add to 64-Bit)	417
100.	UMAQAA (Unsigned Multiply Four Bytes with 32-	419
101.	UMSR64 (Unsigned Multiply and Subtract from 64-Bit Data)	421
102.	UMUL8, UMULX8	423
103.	UMUL16, UMULX16	426
104.	URADD64 (64-bit Unsigned Averaging Addition)	429
105.	URCRAS16 (SIMD 16-bit Unsigned Averaging Cross Addition & Subtraction)	431
106.	URRSA16 (SIMD 16-bit Unsigned Averaging)	433
107.	URSTAS16 (SIMD 16-bit Unsigned Averaging)	435
108.	URSTA16 (SIMD 16-bit Unsigned Averaging)	437
109.	URSUB64 (64-bit Unsigned Averaging Subtraction)	439
110.	WEXTI (Extract Word from 64-bit Immediate)	441
111.	WEXT (Extract Word from 64-bit)	443
112.	ZUNPKD810, ZUNPKD820, ZUNPKD830,	445
113.	KDMBB16, KDMBT16, KDMTT16	448
114.	KDMABB16, KDMABT16, KDMATT16	451
115.	KHMBB16, KHMBT16, KHMTT16	454
116.	KMSDA32, KMSXDA32	457
117.	KSLL32 (SIMD 32-bit Saturating Shift Left)	459
118.	KSLLI32 (SIMD 32-bit Saturating Shift Left Logical)	461
119.	KSLRA32, KSLRA32.u	463
120.	KSTAS32 (SIMD 32-bit Signed Saturating Straight)	465
121.	KSTA32 (SIMD 32-bit Signed Saturating Straight)	467
122.	PKBB32, PKBT32, PKTT32, PKTB32	469
123.	RSTAS32 (SIMD 32-bit Signed Averaging Straight Addition & Subtraction)	472
124.	RSTA32 (SIMD 32-bit Signed Averaging Straight)	473
125.	SLL32 (SIMD 32-bit Shift Left Logical)	474
126.	SLLI32 (SIMD 32-bit Shift Left Logical Immediate)	476
127.	SRA32, SRA32.u	478
128.	SRAI32, SRAI32.u	481
129.	SRAIW.u (Rounding Shift Right Arithmetic)	484
130.	SRL32, SRL32.u	486
131.	SRLI32, SRLI32.u	488

132. STAS32 (SIMD 32-bit Straight Addition &.....	490
133. STSA32 (SIMD 32-bit Straight Subtraction &.....	491
134. UKCRASS32 (SIMD 32-bit Unsigned Saturating.....	493
135. UKCRSA32 (SIMD 32-bit Unsigned Saturating.....	495
136. UKSTAS32 (SIMD 32-bit Unsigned Saturating.....	497
137. UKSTA32 (SIMD 32-bit Unsigned Saturating .....	499
138. URCRAS32 (SIMD 32-bit Unsigned Averaging Cross Addition & Subtraction) .....	501
139. URCRSA32 (SIMD 32-bit Unsigned Averaging.....	568
140. URSTAS32 (SIMD 32-bit Unsigned Averaging.....	570
141. URSTA32 (SIMD 32-bit Unsigned Averaging .....	571

# Revision History

<b>Rev.</b>	<b>Revision Date</b>	<b>Author</b>	<b>Revised Content</b>
v0.01.01	2023/07/27	Rich Fuhler	<ul style="list-style-type: none"><li>• Changed mnemonics based on RVP-baseinstrs-A-002 and update terminology</li><li>• Moved all instructions not included above, which are still up for discussion, into new appendix B</li><li>• Grayed out instructions that are still under discussion</li><li>• Hid chapters 4, 5, 6, 11, and appendix A</li><li>• Hid all C/C++ intrinsic functions for now</li></ul>

# 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. Shorthand Definitions and Terminology

## 2.1. Shorthand Definitions

- $r.H == r.H1: r[31:16], r.L == r.H0: r[15:0]$
- $r.B3: r[31:24], r.B2: r[23:16], r.B1: r[15:8], r.B0: r[7:0]$
- $r.B[x]: r[(x*8+7):(x*8+0)]$
- $r.H[x]: r[(x*16+15):(x*16+0)]$
- $r.W[x]: r[(x*32+31):(x*32+0)]$
- $r.D[x]: r[(x*64+63):(x*64+0)]$
- $r[xU]:$  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.
- $u<<:$  logical left shift, shifting in 0 from the right side.
- $SAT.Qn():$  Saturate to the range of  $[-2^n, 2^n-1]$ , if saturation happens, set OV flag.
- $SAT.Um():$  Saturate to the range of  $[0, 2^m-1]$ , if saturation happens, set OV flag.
- $ROUND():$  Indicate “rounding”, i.e., add 1 to the most significant discarded bit for right shift or MSW-type multiplication instructions.
- $SUM():$  Summation of all data elements.
- Sign or Zero Extending functions:
  - $SE_m(data):$  Sign-Extend data to  $m$ -bit.
  - $SE_{XLEN}(data):$  Sign-Extend data to  $XLEN$ -bit.
  - $ZE_m(data):$  Zero-Extend data to  $m$ -bit.
  - $ZE_{XLEN}(data):$  Zero-Extend data to  $XLEN$ -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 less than comparison.

- $s \leq$ : Signed less than & equal comparison.
- $s >$ : Signed greater than comparison.
- $s^*$ : Signed multiplication.
- $u^*$ : Unsigned multiplication.
- $su^*$ : Signed and Unsigned multiplication.

## 2.2. Terminology

- GPR: General purpose register.
- Q-format ( $Qm.n$ ): It describes a signed binary fixed point number format. "m" is the number of bits, including the sign bit and integer bits, before a notional binary point, and "n" is the number of fraction bits that follow it. This notation represents a signed binary fixed point value in the range of  $-2^{(m-1)}$  (inclusive) and  $2^{(m-1)}$  (exclusive), with  $2^{(m+n)}$  unique values available in that range. For example, Q1.15 represents a number in the range of -1 (inclusive) and 1 (exclusive), with 65536 unique values available in that range.
- Qn: A shorthand format for  $Q1.n$ . For example, Q7, Q15, Q31, Q63.
- Um: It represents an unsigned binary number in the range of 0 and  $(2^m)-1$ .

# Chapter 3. RISC-V P Extension Instruction Summary

## 3.1. SIMD Data Processing Instructions

### 3.1.1. 16-bit Addition & Subtraction Instructions

Based on the combination of the types of the two 16-bit arithmetic operations within a 32-bit word element, 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 Averaging (keeping overflow by dropping 1 LSB bit), Unsigned Averaging, 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	PADD.H (ADD16) rd, rs1, rs2	16-bit Addition	$rd.H[x] = rs1.H[x] + rs2.H[x];$ (RV32: x=1..0, RV64: x=3..0)
2	PAADD.H (RADD16) rd, rs1, rs2	16-bit Signed Averaging Addition	$a17[x] = SE17(rs1.H[x]);$ $b17[x] = SE17(rs2.H[x]);$ $t17[x] = a17[x] + b17[x];$ $rd.H[x] = t17[x] \text{ s} \gg 1;$ (RV32: x=1..0, RV64: x=3..0)
3	PAADDU.H (URADD16) rd, rs1, rs2	16-bit Unsigned Averaging Addition	$a17[x] = ZE17(rs1.H[x]);$ $b17[x] = ZE17(rs2.H[x]);$ $t17[x] = a17[x] + b17[x];$ $rd.H[x] = t17[x] \text{ u} \gg 1;$ (RV32: x=1..0, RV64: x=3..0)

No.	Mnemonic	Instruction	Operation
4	<b>PSADD.H (KADD16)</b> rd, rs1, rs2	16-bit Signed Saturating Addition	$a17[x] = SE17(rs1.H[x]);$ $b17[x] = SE17(rs2.H[x]);$ $t17[x] = a17[x] + b17[x];$ $rd.H[x] = SAT.Q15(t17[x]);$  (RV32: x=1..0, RV64: x=3..0)
5	<b>PSADDU.H</b> <b>(UKADD16)</b> rd, rs1, rs2	16-bit Unsigned Saturating Addition	$a17[x] = ZE17(rs1.H[x]);$ $b17[x] = ZE17(rs2.H[x]);$ $t17[x] = a17[x] + b17[x];$ $rd.H[x] = SAT.U16(t17[x]);$  (RV32: x=1..0, RV64: x=3..0)
6	<b>PSUB.H (SUB16)</b> rd, rs1, rs2	16-bit Subtraction	$rd.H[x] = rs1.H[x] - rs2.H[x];$  (RV32: x=1..0, RV64: x=3..0)
7	<b>PASUB.H (RSUB16)</b> rd, rs1, rs2	16-bit Signed Averaging Subtraction	$a17[x] = SE17(rs1.H[x]);$ $b17[x] = SE17(rs2.H[x]);$ $t17[x] = a17[x] - b17[x];$ $rd.H[x] = t17[x] \text{ s} >> 1;$  (RV32: x=1..0, RV64: x=3..0)
8	<b>PASUBU.H</b> <b>(URSUB16)</b> rd, rs1, rs2	16-bit Unsigned Averaging Subtraction	$a17[x] = ZE17(rs1.H[x]);$ $b17[x] = ZE17(rs2.H[x]);$ $t17[x] = a17[x] - b17[x];$ $rd.H[x] = t17[x] \text{ u} >> 1;$  (RV32: x=1..0, RV64: x=3..0)
9	<b>PSSUB.H (KSUB16)</b> rd, rs1, rs2	16-bit Signed Saturating Subtraction	$a17[x] = SE17(rs1.H[x]);$ $b17[x] = SE17(rs2.H[x]);$ $t17[x] = a17[x] - b17[x];$ $rd.H[x] = SAT.Q15(t17[x]);$  (RV32: x=1..0, RV64: x=3..0)

No.	Mnemonic	Instruction	Operation
10	PSSUBU.H (UKSUB16) rd, rs1, rs2	16-bit Unsigned Saturating Subtraction	$a17[x] = ZE17(rs1.H[x]);$ $b17[x] = ZE17(rs2.H[x]);$ $t17[x] = a17[x] - b17[x];$ $rd.H[x] = SAT.U16(t17[x]);$ (RV32: x=1..0, RV64: x=3..0)
11	PAS.HX (CRAS16) rd, rs1, rs2	16-bit Cross Add & Sub	$rd.H[x] = rs1.H[x] + rs2.H[x-1];$ $rd.H[x-1] = rs1.H[x-1] - rs2.H[x];$ (RV32: x=1, RV64: x=1,3)
12	PAAS.HX (RCRAS16) rd, rs1, rs2	16-bit Signed Averaging Cross Add & Sub	$ah17[x] = SE17(rs1.H[x]);$ $bh17[x] = SE17(rs2.H[x]);$ $al17[x] = SE17(rs1.H[x-1]);$ $bl17[x] = SE17(rs2.H[x-1]);$ $e17[x] = ah17[x] + bl17[x];$ $f17[x] = al17[x] - bh17[x];$ $rd.H[x] = e17[x] \text{ s} >> 1;$ $rd.H[x-1] = f17[x] \text{ s} >> 1;$ (RV32: x=1, RV64: x=1,3)
13	URCRAS16 rd, rs1, rs2	16-bit Unsigned Averaging Cross Add & Sub	$ah17[x] = ZE17(rs1.H[x]);$ $bh17[x] = ZE17(rs2.H[x]);$ $al17[x] = ZE17(rs1.H[x-1]);$ $bl17[x] = ZE17(rs2.H[x-1]);$ $th17[x] = ah17[x] + bl17[x];$ $tl17[x] = al17[x] - bh17[x];$ $rd.H[x] = th17[x] \text{ u} >> 1;$ $rd.H[x-1] = tl17[x] \text{ u} >> 1;$ (RV32: x=1, RV64: x=1,3)

No.	Mnemonic	Instruction	Operation
14	PSAS.HX (KCRAS16) rd, rs1, rs2	16-bit Signed Saturating Cross Add & Sub	$\begin{aligned} ah17[x] &= SE17(rs1.H[x]); \\ bh17[x] &= SE17(rs2.H[x]); \\ al17[x] &= SE17(rs1.H[x-1]); \\ bl17[x] &= SE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] + bl17[x]; \\ tl17[x] &= al17[x] - bh17[x]; \\ rd.H[x] &= SAT.Q15(th17[x]); \\ rd.H[x-1] &= SAT.Q15(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
15	UKCRAS16 rd, rs1, rs2	16-bit Unsigned Saturating Cross Add & Sub	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] + bl17[x]; \\ tl17[x] &= al17[x] - bh17[x]; \\ rd.H[x] &= SAT.U16(th17[x]); \\ rd.H[x-1] &= SAT.U16(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
16	PSA.HX (CRSA16) rd, rs1, rs2	16-bit Cross Sub & Add	$\begin{aligned} rd.H[x] &= rs1.H[x] - rs2.H[x-1]; \\ rd.H[x-1] &= rs1.H[x-1] + rs2.H[x]; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
17	PASA.HX (RCRSA16) rd, rs1, rs2	16-bit Signed Averaging Cross Sub & Add	$\begin{aligned} ah17[x] &= SE17(rs1.H[x]); \\ bh17[x] &= SE17(rs2.H[x]); \\ al17[x] &= SE17(rs1.H[x-1]); \\ bl17[x] &= SE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bl17[x]; \\ tl17[x] &= al17[x] + bh17[x]; \\ rd.H[x] &= th17[x] s>> 1; \\ rd.H[x-1] &= tl17[x] s>> 1; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>

No.	Mnemonic	Instruction	Operation
18	URCRSA16 rd, rs1, rs2	16-bit Unsigned Averaging Cross Sub & Add	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bl17[x]; \\ tl17[x] &= al17[x] + bh17[x]; \\ rd.H[x] &= th17[x] \text{ u}>> 1; \\ rd.H[x-1] &= tl17[x] \text{ u}>> 1; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
19	PSSA.HX (KCRSA16) rd, rs1, rs2	16-bit Signed Saturating Cross Sub & Add	$\begin{aligned} ah17[x] &= SE17(rs1.H[x]); \\ bh17[x] &= SE17(rs2.H[x]); \\ al17[x] &= SE17(rs1.H[x-1]); \\ bl17[x] &= SE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bl17[x]; \\ tl17[x] &= al17[x] + bh17[x]; \\ rd.H[x] &= SAT.Q15(th17[x]); \\ rd.H[x-1] &= SAT.Q15(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
20	UKCRSA16 rd, rs1, rs2	16-bit Unsigned Saturating Cross Sub & Add	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bl17[x]; \\ tl17[x] &= al17[x] + bh17[x]; \\ rd.H[x] &= SAT.U16(th17[x]); \\ rd.H[x-1] &= SAT.U16(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
21	STAS16 rd, rs1, rs2	16-bit Straight Add & Sub	$\begin{aligned} rd.H[x] &= rs1.H[x] + rs2.H[x]; \\ rd.H[x-1] &= rs1.H[x-1] - rs2.H[x-1]; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>

No.	Mnemonic	Instruction	Operation
22	RSTAS16 rd, rs1, rs2	16-bit Signed Averaging Straight Add & Sub	$  \begin{aligned}  ah17[x] &= SE17(rs1.H[x]); \\  bh17[x] &= SE17(rs2.H[x]); \\  al17[x] &= SE17(rs1.H[x-1]); \\  bl17[x] &= SE17(rs2.H[x-1]); \\  th17[x] &= ah17[x] + bh17[x]; \\  tl17[x] &= al17[x] - bl17[x]; \\  rd.H[x] &= th17[x] s>> 1; \\  rd.H[x-1] &= tl17[x] s>> 1;  \end{aligned}  $ <p>(RV32: x=1, RV64: x=1,3)</p>
23	URSTAS16 rd, rs1, rs2	16-bit Unsigned Averaging Straight Add & Sub	$  \begin{aligned}  ah17[x] &= ZE17(rs1.H[x]); \\  bh17[x] &= ZE17(rs2.H[x]); \\  al17[x] &= ZE17(rs1.H[x-1]); \\  bl17[x] &= ZE17(rs2.H[x-1]); \\  th17[x] &= ah17[x] + bh17[x]; \\  tl17[x] &= al17[x] - bl17[x]; \\  rd.H[x] &= th17[x] u>> 1; \\  rd.H[x-1] &= tl17[x] u>> 1;  \end{aligned}  $ <p>(RV32: x=1, RV64: x=1,3)</p>
24	KSTAS16 rd, rs1, rs2	16-bit Signed Saturating Straight Add & Sub	$  \begin{aligned}  ah17[x] &= SE17(rs1.H[x]); \\  bh17[x] &= SE17(rs2.H[x]); \\  al17[x] &= SE17(rs1.H[x-1]); \\  bl17[x] &= SE17(rs2.H[x-1]); \\  th17[x] &= ah17[x] + bh17[x]; \\  tl17[x] &= al17[x] - bl17[x]; \\  rd.H[x] &= SAT.Q15(th17[x]); \\  rd.H[x-1] &= SAT.Q15(tl17[x]);  \end{aligned}  $ <p>(RV32: x=1, RV64: x=1,3)</p>

No.	Mnemonic	Instruction	Operation
25	UKSTAS16 rd, rs1, rs2	16-bit Unsigned Saturating Straight Add & Sub	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] + bh17[x]; \\ tl17[x] &= al17[x] - bl17[x]; \\ rd.H[x] &= SAT.U16(th17[x]); \\ rd.H[x-1] &= SAT.U16(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
26	STSA16 rd, rs1, rs2	16-bit Straight Sub & Add	$\begin{aligned} rd.H[x] &= rs1.H[x] - rs2.H[x]; \\ rd.H[x-1] &= rs1.H[x-1] + rs2.H[x-1]; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
27	RSTSA16 rd, rs1, rs2	16-bit Signed Averaging Straight Sub & Add	$\begin{aligned} ah17[x] &= SE17(rs1.H[x]); \\ bh17[x] &= SE17(rs2.H[x]); \\ al17[x] &= SE17(rs1.H[x-1]); \\ bl17[x] &= SE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bh17[x]; \\ tl17[x] &= al17[x] + bl17[x]; \\ rd.H[x] &= th17[x] s\gg 1; \\ rd.H[x-1] &= tl17[x] s\gg 1; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
28	URSTSA16 rd, rs1, rs2	16-bit Unsigned Averaging Straight Sub & Add	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bh17[x]; \\ tl17[x] &= al17[x] + bl17[x]; \\ rd.H[x] &= th17[x] u\gg 1; \\ rd.H[x-1] &= tl17[x] u\gg 1; \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>

No.	Mnemonic	Instruction	Operation
29	KSTSA16 rd, rs1, rs2	16-bit Signed Saturating Straight Sub & Add	$\begin{aligned} ah17[x] &= SE17(rs1.H[x]); \\ bh17[x] &= SE17(rs2.H[x]); \\ al17[x] &= SE17(rs1.H[x-1]); \\ bl17[x] &= SE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bh17[x]; \\ tl17[x] &= al17[x] + bl17[x]; \\ rd.H[x] &= SAT.Q15(th17[x]); \\ rd.H[x-1] &= SAT.Q15(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>
30	UKSTSA16 rd, rs1, rs2	16-bit Unsigned Saturating Straight Sub & Add	$\begin{aligned} ah17[x] &= ZE17(rs1.H[x]); \\ bh17[x] &= ZE17(rs2.H[x]); \\ al17[x] &= ZE17(rs1.H[x-1]); \\ bl17[x] &= ZE17(rs2.H[x-1]); \\ th17[x] &= ah17[x] - bh17[x]; \\ tl17[x] &= al17[x] + bl17[x]; \\ rd.H[x] &= SAT.U16(th17[x]); \\ rd.H[x-1] &= SAT.U16(tl17[x]); \end{aligned}$ <p>(RV32: x=1, RV64: x=1,3)</p>

### 3.1.2. 8-bit Addition & Subtraction Instructions

Based on the types of the four 8-bit arithmetic operations within a 32-bit word element, 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 signed or unsigned operation, the SIMD 8-bit add/subtract instructions can be classified into 5 groups: Wrap-around (dropping overflow), Signed Averaging (keeping overflow by dropping 1 LSB bit), Unsigned Averaging, 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	PADD.B rd, rs1, rs2	8-bit Addition	$rd.B[x] = rs1.B[x] + rs2.B[x];$ $(RV32: x=3..0, RV64: x=7..0)$
2	PAADD.B (RADD8) rd, rs1, rs2	8-bit Signed Averaging Addition	$a9[x] = SE9(rs1.B[x]);$ $b9[x] = SE9(rs2.B[x]);$ $t9[x] = a9[x] + b9[x];$ $rd.B[x] = t9[x] \text{ s} >> 1;$ $(RV32: x=3..0, RV64: x=7..0)$
3	PAADDU.B (URADD8) rd, rs1, rs2	8-bit Unsigned Averaging Addition	$a9[x] = ZE9(rs1.B[x]);$ $b9[x] = ZE9(rs2.B[x]);$ $rd.B[x] = (a9[x] + b9[x]) \text{ u} >> 1;$ $(RV32: x=3..0, RV64: x=7..0)$
4	KADD8 rd, rs1, rs2	8-bit Signed Saturating Addition	$a9[x] = SE9(rs1.B[x]);$ $b9[x] = SE9(rs2.B[x]);$ $t9[x] = a9[x] + b9[x];$ $rd.B[x] = SAT.Q7(t9[x]);$ $(RV32: x=3..0, RV64: x=7..0)$

No.	Mnemonic	Instruction	Operation
5	<b>PSADDU.B</b> <b>(UKADD8)</b> rd, rs1, rs2	8-bit Unsigned Saturating Addition	$a9[x] = ZE9(rs1.B[x]);$ $b9[x] = ZE9(rs2.B[x]);$ $t9[x] = a9[x] + b9[x];$ $rd.H[x] = SAT.U8(t9[x]);$  (RV32: x=1..0, RV64: x=3..0)
6	<b>PSUB.B</b> ( <b>SUB8</b> ) rd, rs1, rs2	8-bit Subtraction	$rd.B[x] = rs1.B[x] - rs2.B[x];$  (RV32: x=3..0, RV64: x=7..0)
7	<b>PASUB.B</b> ( <b>RSUB8</b> ) rd, rs1, rs2	8-bit Signed Averaging Subtraction	$a9[x] = SE9(rs1.B[x]);$ $b9[x] = SE9(rs2.B[x]);$ $t9[x] = a9[x] - b9[x];$ $rd.B[x] = t9[x] s\gg 1;$  (RV32: x=3..0, RV64: x=7..0)
8	<b>PASUBU.B</b> ( <b>URSUB8</b> ) rd, rs1, rs2	8-bit Unsigned Averaging Subtraction	$a9[x] = ZE9(rs1.B[x]);$ $b9[x] = ZE9(rs2.B[x]);$ $rd.B[x] = (a9[x] - b9[x]) u\gg 1;$  (RV32: x=3..0, RV64: x=7..0)
9	<b>PSSUB.B</b> ( <b>KSUB8</b> ) rd, rs1, rs2	8-bit Signed Saturating Subtraction	$a9[x] = SE9(rs1.B[x]);$ $b9[x] = SE9(rs2.B[x]);$ $t9[x] = a9[x] - b9[x];$ $rd.B[x] = SAT.Q7(t9[x]);$  (RV32: x=3..0, RV64: x=7..0)
10	<b>PSSUBU.B</b> ( <b>UKSUB8</b> ) rd, rs1, rs2	8-bit Unsigned Saturating Subtraction	$a9[x] = ZE9(rs1.B[x]);$ $b9[x] = ZE9(rs2.B[x]);$ $t9[x] = a9[x] - b9[x];$ $rd.H[x] = SAT.U8(t9[x]);$  (RV32: x=1..0, RV64: x=3..0)

### 3.1.3. 16-bit Shift Instructions

There are 14 instructions here.

Table 3. SIMD 16-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRA16 rd, rs1, rs2	16-bit Shift Right Arithmetic	$rd.H[x] = rs1.H[x] \text{ s}>> rs2[3:0];$ (RV32: x=1..0, RV64: x=3..0)
2	SRAI16 rd, rs1, im4u	16-bit Shift Right Arithmetic Immediate	$rd.H[x] = rs1.H[x] \text{ s}>> im4u;$ (RV32: x=1..0, RV64: x=3..0)
3	SRA16.u rd, rs1, rs2	16-bit Rounding Shift Right Arithmetic	$a[x] = rs1.H[x];$ $rd.H[x] = \text{ROUND}(a[x] \text{ s}>> rs2[3:0]);$ (RV32: x=1..0, RV64: x=3..0)
4	SRAI16.u rd, rs1, im4u	16-bit Rounding Shift Right Arithmetic Immediate	$rd.H[x] = \text{ROUND}(rs1.H[x] \text{ s}>> im4u);$ (RV32: x=1..0, RV64: x=3..0)
5	SRL16 rd, rs1, rs2	16-bit Shift Right Logical	$rd.H[x] = rs1.H[x] \text{ u}>> rs2[3:0];$ (RV32: x=1..0, RV64: x=3..0)
6	SRLI16 rd, rs1, im4u	16-bit Shift Right Logical Immediate	$rd.H[x] = rs1.H[x] \text{ u}>> im4u;$ (RV32: x=1..0, RV64: x=3..0)
7	SRL16.u rd, rs1, rs2	16-bit Rounding Shift Right Logical	$a[x] = rs1.H[x];$ $rd.H[x] = \text{ROUND}(a[x] \text{ u}>> rs2[3:0]);$ (RV32: x=1..0, RV64: x=3..0)

No.	Mnemonic	Instruction	Operation
8	SRLI16.u rd, rs1, im4u	16-bit Rounding Shift Right Logical Immediate	$rd.H[x] = \text{ROUND}(rs1.H[x] \text{ u}>> \text{im4u});$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
9	SLL16 rd, rs1, rs2	16-bit Shift Left Logical	$rd.H[x] = rs1.H[x] \ll rs2[3:0];$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
10	SLLI16 rd, rs1, im4u	16-bit Shift Left Logical Immediate	$rd.H[x] = rs1.H[x] \ll \text{im4u};$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
11	KSLL16 rd, rs1, rs2	16-bit Saturating Shift Left Logical	$a[x] = rs1.H[x];$ $rd.H[x] = \text{SAT.Q15}(a[x] \ll rs2[3:0]);$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
12	KSLLI16 rd, rs1, im4u	16-bit Saturating Shift Left Logical Immediate	$rd.H[x] = \text{SAT.Q15}(rs1.H[x] \ll \text{im4u});$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
13	KSLRA16 rd, rs1, rs2	16-bit Shift Left Logical with Saturation & Shift Right Arithmetic	$a[x] = rs1.H[x];$ $\text{if } (rs2[4:0] \leq 0)$ $\quad t[x] = a[x] \text{ s}>> -rs2[4:0];$ $\text{if } (rs2[4:0] \geq 0)$ $\quad t[x] = \text{SAT.Q15}(a[x] \ll rs2[4:0]);$ $rd.H[x] = t[x];$  (RV32: $x=1..0$ , RV64: $x=3..0$ )

No.	Mnemonic	Instruction	Operation
14	KSLRA16.u rd, rs1, rs2	16-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	<pre> a[x] = rs1.H[x]; if (rs2[4:0] s&lt; 0)     t[x] = ROUND(a[x] s&gt;&gt; -rs2[4:0]); if (rs2[4:0] s&gt; 0)     t[x] = SAT.Q15(a[x] &lt;&lt; rs2[4:0]); rd.H[x] = t[x]; </pre> <p>(RV32: x=1..0, RV64: x=3..0)</p>

### 3.1.4. 8-bit Shift Instructions

There are 14 instructions here.

Table 4. SIMD 8-bit Shift Instructions

No.	Mnemonic	Instruction	Operation
1	SRA8 rd, rs1, rs2	8-bit Shift Right Arithmetic	$rd.B[x] = rs1.B[x] \text{ s}>> rs2[2:0];$ (RV32: x=3..0, RV64: x=7..0)
2	SRAI8 rd, rs1, im3u	8-bit Shift Right Arithmetic Immediate	$rd.B[x] = rs1.B[x] \text{ s}>> im3u;$ (RV32: x=3..0, RV64: x=7..0)
3	SRA8.u rd, rs1, rs2	8-bit Rounding Shift Right Arithmetic	$a[x] = rs1.B[x];$ $rd.B[x] = \text{ROUND}(a[x] \text{ s}>> rs2[2:0]);$ (RV32: x=3..0, RV64: x=7..0)
4	SRAI8.u rd, rs1, im3u	8-bit Rounding Shift Right Arithmetic Immediate	$rd.B[x] = \text{ROUND}(rs1.B[x] \text{ s}>> im3u);$ (RV32: x=3..0, RV64: x=7..0)
5	SRL8 rd, rs1, rs2	8-bit Shift Right Logical	$rd.B[x] = rs1.B[x] \text{ u}>> rs2[2:0];$ (RV32: x=3..0, RV64: x=7..0)
6	SRLI8 rd, rs1, im3u	8-bit Shift Right Logical Immediate	$rd.B[x] = rs1.B[x] \text{ u}>> im3u;$ (RV32: x=3..0, RV64: x=7..0)
7	SRL8.u rd, rs1, rs2	8-bit Rounding Shift Right Logical	$a[x] = rs1.B[x];$ $rd.B[x] = \text{ROUND}(a[x] \text{ u}>> rs2[2:0]);$ (RV32: x=3..0, RV64: x=7..0)

No.	Mnemonic	Instruction	Operation
8	SRLI8.u rd, rs1, im3u	8-bit Rounding Shift Right Logical Immediate	$rd.B[x] = \text{ROUND}(rs1.B[x] \text{ u}>> \text{im3u});$ (RV32: x=3..0, RV64: x=7..0)
9	SLL8 rd, rs1, rs2	8-bit Shift Left Logical	$rd.B[x] = rs1.B[x] \ll rs2[2:0];$ (RV32: x=3..0, RV64: x=7..0)
10	SLLI8 rd, rs1, im3u	8-bit Shift Left Logical Immediate	$rd.B[x] = rs1.B[x] \ll \text{im3u};$ (RV32: x=3..0, RV64: x=7..0)
11	KSLL8 rd, rs1, rs2	8-bit Saturating Shift Left Logical	$a[x] = rs1.B[x];$ $rd.B[x] = \text{SAT.Q7}(a[x] \ll rs2[2:0]);$ (RV32: x=3..0, RV64: x=7..0)
12	KSLLI8 rd, rs1, im3u	8-bit Saturating Shift Left Logical Immediate	$rd.B[x] = \text{SAT.Q7}(rs1.B[x] \ll \text{im3u});$ (RV32: x=3..0, RV64: x=7..0)
13	KSLRA8 rd, rs1, rs2	8-bit Shift Left Logical with Saturation & Shift Right Arithmetic	$a[x] = rs1.B[x];$ $\text{if } (rs2[3:0] \leq 0)$ $\quad t[x] = a[x] \text{ s}>> -rs2[3:0];$ $\text{if } (rs2[3:0] > 0)$ $\quad t[x] = \text{SAT.Q7}(a[x] \ll rs2[3:0]);$ $rd.B[x] = t[x];$  (RV32: x=3..0, RV64: x=7..0)

No.	Mnemonic	Instruction	Operation
14	KSLRA8.u rd, rs1, rs2	8-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	<pre> a[x] = rs1.B[x]; if (rs2[3:0] s&lt; 0)     t[x] = ROUND(a[x] s&gt;&gt; -rs2[3:0]); if (rs2[3:0] s&gt; 0)     t[x] = SAT.Q7(a[x] &lt;&lt; rs2[3:0]); rd.B[x] = t[x]; </pre> <p>(RV32: x=3..0, RV64: x=7..0)</p>

### 3.1.5. 16-bit Compare Instructions

There are 5 instructions here.

Table 5. SIMD 16-bit Compare Instructions

No.	Mnemonic	Instruction	Operation
1	<b>PMSEQ.H</b> (CMPEQ16) rd, rs1, rs2	16-bit Compare Equal	$eq[x] = (rs1.H[x] == rs2.H[x]);$ $rd.H[x] = eq[x]? 0xffff : 0;$  (RV32: x=1..0, RV64: x=3..0)
2	<b>PMSLT.H</b> (SCMPLT16) rd, rs1, rs2	16-bit Signed Compare Less Than	$lt[x] = (rs1.H[x] < rs2.H[x]);$ $rd.H[x] = lt[x]? 0xffff : 0;$  (RV32: x=1..0, RV64: x=3..0)
3	<b>PMSGE.H</b> (SCMPLE16) rd, rs1, rs2	16-bit Signed Compare Less Than & Equal	$le[x] = (rs1.H[x] \leq rs2.H[x]);$ $rd.H[x] = le[x]? 0xffff : 0;$  (RV32: x=1..0, RV64: x=3..0)
4	<b>PMSLTU.H</b> (UCMPLT16) rd, rs1, rs2	16-bit Unsigned Compare Less Than	$ult[x] = (rs1.H[x] < rs2.H[x]);$ $rd.H[x] = ult[x]? 0xffff : 0;$  (RV32: x=1..0, RV64: x=3..0)
5	<b>PMSGEU.H</b> (UCMPLE16) rd, rs1, rs2	16-bit Unsigned Compare Less Than & Equal	$uge[x] = (rs1.H[x] \geq rs2.H[x]);$ $rd.H[x] = uge[x]? 0xffff : 0;$  (RV32: x=1..0, RV64: x=3..0)

### 3.1.6. 8-bit Compare Instructions

There are 5 instructions here.

Table 6. SIMD 8-bit Compare Instructions

No.	Mnemonic	Instruction	Operation
1	<b>PMSEQ.B (CMPEQ8)</b> rd, rs1, rs2	8-bit Compare Equal	$\text{eq}[x] = (\text{rs1.B}[x] == \text{rs2.B}[x]);$ $\text{rd.B}[x] = \text{eq}[x]? 0xff : 0;$ (RV32: x=3..0, RV64: x=7..0)
2	<b>PMSLT.B (SCMPLT8)</b> rd, rs1, rs2	8-bit Signed Compare Less Than	$\text{lt}[x] = (\text{rs1.B}[x] < \text{rs2.B}[x]);$ $\text{rd.B}[x] = \text{lt}[x]? 0xff : 0;$ (RV32: x=3..0, RV64: x=7..0)
3	<b>PMSGE.B (SCMPLE8)</b> rd, rs1, rs2	8-bit Signed Compare Greater Than or Equal	$\text{ge}[x] = (\text{rs1.B}[x] \geq \text{rs2.B}[x]);$ $\text{rd.B}[x] = \text{ge}[x]? 0xff : 0;$ (RV32: x=3..0, RV64: x=7..0)
4	<b>PMSLTU.B (UCMPLT8)</b> rd, rs1, rs2	8-bit Unsigned Compare Less Than	$\text{ult}[x] = (\text{rs1.B}[x] < \text{rs2.B}[x]);$ $\text{rd.B}[x] = \text{ult}[x]? 0xff : 0;$ (RV32: x=3..0, RV64: x=7..0)
5	<b>PMSGEU.B (UCMPLUE8)</b> rd, rs1, rs2	8-bit Unsigned Compare Greater Than or Equal	$\text{ule}[x] = (\text{rs1.B}[x] \geq \text{rs2.B}[x]);$ $\text{rd.B}[x] = \text{ule}[x]? 0xff : 0;$ (RV32: x=3..0, RV64: x=7..0)

### 3.1.7. 16-bit Multiply Instructions

There are 6 instructions here.

Table 7. SIMD 16-bit Multiply Instructions

No.	Mnemonic	Instruction	Operation
1	SMUL16 rd, rs1, rs2	16-bit Signed Multiply	<p>RV32:</p> $r[dL] = rs1.H[0] s^* rs2.H[0];$ $r[dU] = rs1.H[1] s^* rs2.H[1];$ <p>RV64:</p> $rd.W[0] = rs1.H[0] s^* rs2.H[0];$ $rd.W[1] = rs1.H[1] s^* rs2.H[1];$
2	SMULX16 rd, rs1, rs2	16-bit Signed Crossed Multiply	<p>RV32:</p> $r[dL] = rs1.H[0] s^* rs2.H[1];$ $r[dU] = rs1.H[1] s^* rs2.H[0];$ <p>RV64:</p> $rd.W[0] = rs1.H[0] s^* rs2.H[1];$ $rd.W[1] = rs1.H[1] s^* rs2.H[0];$
3	UMUL16 rd, rs1, rs2	16-bit Unsigned Multiply	<p>RV32:</p> $r[dL] = rs1.H[0] u^* rs2.H[0];$ $r[dU] = rs1.H[1] u^* rs2.H[1];$ <p>RV64:</p> $rd.W[0] = rs1.H[0] u^* rs2.H[0];$ $rd.W[1] = rs1.H[1] u^* rs2.H[1];$

No.	Mnemonic	Instruction	Operation
4	UMULX16 rd, rs1, rs2	16-bit Unsigned Crossed Multiply	<p>RV32:</p> $r[dL] = rs1.H[0] \text{ u}^* rs2.H[1];$ $r[dU] = rs1.H[1] \text{ u}^* rs2.H[0];$ <p>RV64:</p> $rd.W[0] = rs1.H[0] \text{ u}^* rs2.H[1];$ $rd.W[1] = rs1.H[1] \text{ u}^* rs2.H[0];$
5	PMULQ.H (KHM16) rd, rs1, rs2	Q15 Signed Saturating Multiply	$t[x] = rs1.H[x] s^* rs2.H[x];$ $rd.H[x] = \text{SAT.Q15}(t[x] s\gg 15);$ <p>(RV32: x=1..0, RV64: x=3..0)</p>
6	KHMX16 rd, rs1, rs2	Q15 Signed Saturating Crossed Multiply	$t[x] = rs1.H[x] s^* rs2.H[y];$ $rd.H[x] = \text{SAT.Q15}(t[x] s\gg 15);$ <p>(RV32: (x,y)=(1,0),(0,1), RV64: (x,y)=(3,2),(2,3), (1,0),(0,1))</p>

### 3.1.8. 8-bit Multiply Instructions

There are 6 instructions here.

*Table 8. SIMD 8-bit Multiply Instructions*

No.	Mnemonic	Instruction	Operation
1	SMUL8 rd, rs1, rs2	8-bit Signed Multiply	<p>RV32:</p> $r[dL].H[0] = rs1.B[0] s^* rs2.B[0];$ $r[dL].H[1] = rs1.B[1] s^* rs2.B[1];$ $r[dU].H[0] = rs1.B[2] s^* rs2.B[2];$ $r[dU].H[1] = rs1.B[3] s^* rs2.B[3];$ <p>RV64:</p> $rd.H[0] = rs1.B[0] s^* rs2.B[0];$ $rd.H[1] = rs1.B[1] s^* rs2.B[1];$ $rd.H[2] = rs1.B[2] s^* rs2.B[2];$ $rd.H[3] = rs1.B[3] s^* rs2.B[3];$
2	SMULX8 rd, rs1, rs2	8-bit Signed Crossed Multiply	<p>RV32:</p> $r[dL].H[0] = rs1.B[0] s^* rs2.B[1];$ $r[dL].H[1] = rs1.B[1] s^* rs2.B[0];$ $r[dU].H[0] = rs1.B[2] s^* rs2.B[3];$ $r[dU].H[1] = rs1.B[3] s^* rs2.B[2];$ <p>RV64:</p> $rd.H[0] = rs1.B[0] s^* rs2.B[1];$ $rd.H[1] = rs1.B[1] s^* rs2.B[0];$ $rd.H[2] = rs1.B[2] s^* rs2.B[3];$ $rd.H[3] = rs1.B[3] s^* rs2.B[2];$

No.	Mnemonic	Instruction	Operation
3	UMUL8 rd, rs1, rs2	8-bit Unsigned Multiply	<p>RV32:</p> $r[dL].H[0] = rs1.B[0] \text{ u}^* rs2.B[0];$ $r[dL].H[1] = rs1.B[1] \text{ u}^* rs2.B[1];$ $r[dU].H[0] = rs1.B[2] \text{ u}^* rs2.B[2];$ $r[dU].H[1] = rs1.B[3] \text{ u}^* rs2.B[3];$ <p>RV64:</p> $rd.H[0] = rs1.B[0] \text{ u}^* rs2.B[0];$ $rd.H[1] = rs1.B[1] \text{ u}^* rs2.B[1];$ $rd.H[2] = rs1.B[2] \text{ u}^* rs2.B[2];$ $rd.H[3] = rs1.B[3] \text{ u}^* rs2.B[3];$
4	UMULX8 rd, rs1, rs2	8-bit Unsigned Crossed Multiply	<p>RV32:</p> $r[dL].H[0] = rs1.B[0] \text{ u}^* rs2.B[1];$ $r[dL].H[1] = rs1.B[1] \text{ u}^* rs2.B[0];$ $r[dU].H[0] = rs1.B[2] \text{ u}^* rs2.B[3];$ $r[dU].H[1] = rs1.B[3] \text{ u}^* rs2.B[2];$ <p>RV64:</p> $rd.H[0] = rs1.B[0] \text{ u}^* rs2.B[1];$ $rd.H[1] = rs1.B[1] \text{ u}^* rs2.B[0];$ $rd.H[2] = rs1.B[2] \text{ u}^* rs2.B[3];$ $rd.H[3] = rs1.B[3] \text{ u}^* rs2.B[2];$
5	KHM8 rd, rs1, rs2	Q7 Signed Saturating Multiply	$t[x] = rs1.B[x] s^* rs2.B[x];$ $rd.B[x] = \text{SAT.Q7}(t[x] s\gg 7);$ <p>(RV32: x=3..0, RV64: x=7..0)</p>
6	KHMX8 rd, rs1, rs2	Q7 Signed Saturating Crossed Multiply	$t[x] = rs1.B[x] s^* rs2.B[y];$ $rd.B[x] = \text{SAT.Q7}(t[x] s\gg 7);$ <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>

### 3.1.9. 16-bit Misc Instructions

There are 11 instructions here.

Table 9. SIMD 16-bit Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	<b>PMIN.H (SMIN16)</b> rd, rs1, rs2	16-bit Signed Minimum	$le[x] = rs1.H[x] \ s < rs2.H[x];$ $rd.H[x] = le[x] ? rs1.H[x] : rs2.H[x];$ (RV32: x=1..0, RV64: x=3..0)
2	<b>PMINU.H (UMIN16)</b> rd, rs1, rs2	16-bit Unsigned Minimum	$le[x] = rs1.H[x] \ u < rs2.H[x];$ $rd.H[x] = le[x] ? rs1.H[x] : rs2.H[x];$ (RV32: x=1..0, RV64: x=3..0)
3	<b>PMAX.H (SMAX16)</b> rd, rs1, rs2	16-bit Signed Maximum	$ge[x] = rs1.H[x] \ s > rs2.H[x];$ $rd.H[x] = ge[x] ? rs1.H[x] : rs2.H[x];$ (RV32: x=1..0, RV64: x=3..0)
4	<b>PMAXU.H (UMAX16)</b> rd, rs1, rs2	16-bit Unsigned Maximum	$ge[x] = rs1.H[x] \ u > rs2.H[x];$ $rd.H[x] = ge[x] ? rs1.H[x] : rs2.H[x];$ (RV32: x=1..0, RV64: x=3..0)
5	SCLIP16 rd, rs1, imm4u	16-bit Signed Clip Value	$n = imm4u;$ $rd.H[x] = SAT.Qn(rs1.H[x]);$ (RV32: x=1..0, RV64: x=3..0)
6	UCLIP16 rd, rs1, imm4u	16-bit Unsigned Clip Value	$m = imm4u;$ $rd.H[x] = SAT.Um(rs1.H[x]);$ (RV32: x=1..0, RV64: x=3..0)

No.	Mnemonic	Instruction	Operation
7	PABS.H (KABS16) rd, rs1	16-bit Absolute Value	$rd.H[x] = ABS(rs1.H[x]);$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
8	CLRS16 rd, rs1	16-bit Count Leading Redundant Sign	$rd.H[x] = CLRS(rs1.H[x]);$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
9	CLZ16 rd, rs1	16-bit Count Leading Zero	$rd.H[x] = CLZ(rs1.H[x]);$ (RV32: $x=1..0$ , RV64: $x=3..0$ )
10	SWAP16 rd, rs1	Swap Halfword within Word	$ah0[x] = rs1.W[x].H[0];$ $ah1[x] = rs1.W[x].H[1];$ $rd.W[x] = CONCAT(ah0[x], ah1[x]);$ (RV32: $x=0$ , RV64: $x=1..0$ )

### 3.1.10. 8-bit Misc Instructions

There are 11 instructions here.

Table 10. SIMD 8-bit Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	<b>PMIN.B (SMIN8) rd, rs1, rs2</b>	8-bit Signed Minimum	$le[x] = rs1.B[x] \ s < rs2.B[x];$ $rd.B[x] = le[x] ? rs1.B[x] : rs2.B[x];$ (RV32: x=3..0, RV64: x=7..0)
2	<b>PSMINU.B (UMIN8) rd, rs1, rs2</b>	8-bit Unsigned Minimum	$le[x] = rs1.B[x] \ u < rs2.B[x];$ $rd.B[x] = le[x] ? rs1.B[x] : rs2.B[x];$ (RV32: x=3..0, RV64: x=7..0)
3	<b>PMAX.B (SMAX8) rd, rs1, rs2</b>	8-bit Signed Maximum	$ge[x] = rs1.B[x] \ s > rs2.B[x];$ $rd.B[x] = ge[x] ? rs1.B[x] : rs2.B[x];$ (RV32: x=3..0, RV64: x=7..0)
4	<b>PMAXU.B (UMAX8) rd, rs1, rs2</b>	8-bit Unsigned Maximum	$ge[x] = rs1.B[x] \ u > rs2.B[x];$ $rd.B[x] = ge[x] ? rs1.B[x] : rs2.B[x];$ (RV32: x=3..0, RV64: x=7..0)
5	KABS8 rd, rs1	8-bit Absolute Value	$rd.B[x] = SAT.Q7(ABS(rs1.B[x]));$ (RV32: x=3..0, RV64: x=7..0)
6	SCLIP8 rd, rs1, imm3u	8-bit Signed Clip Value	$n = imm3u;$ $rd.B[x] = SAT.Qn(rs1.B[x]);$ (RV32: x=3..0, RV64: x=7..0)

No.	Mnemonic	Instruction	Operation
7	UCLIP8 rd, rs1, imm3u	8-bit Unsigned Clip Value	$m = \text{imm3u};$ $\text{rd.B}[x] = \text{SAT.Um}(\text{rs1.B}[x]);$  (RV32: x=3..0, RV64: x=7..0)
8	CLRS8 rd, rs1	8-bit Count Leading Redundant Sign	$\text{rd.B}[x] = \text{CLRS}(\text{rs1.B}[x]);$  (RV32: x=3..0, RV64: x=7..0)
9	CLZ8 rd, rs1	8-bit Count Leading Zero	$\text{rd.B}[x] = \text{CLZ}(\text{rs1.B}[x]);$  (RV32: x=3..0, RV64: x=7..0)
10	SWAP8 rd, rs1	Swap Byte within Halfword	$\text{ab0}[x] = \text{rs1.H}[x].\text{B}[0];$ $\text{ab1}[x] = \text{rs1.H}[x].\text{B}[1];$ $\text{rd.H}[x] = \text{CONCAT}(\text{ab0}[x], \text{ab1}[x]);$  (RV32: x=1..0, RV64: x=3..0)

### 3.1.11. 8-bit Unpacking Instructions

There are 10 instructions here.

*Table 11. 8-bit Unpacking Instructions*

No.	Mnemonic	Instruction	Operation
1	SUNPKD810 rd, rs1	Signed Unpacking Bytes 1 & 0	$rd.H[x] = SE16(rs1.B[y]);$  RV32: $(x,y) = (1,1),(0,0)$ RV64: $(x,y) = (3,5),(2,4), (1,1),(0,0)$
2	SUNPKD820 rd, rs1	Signed Unpacking Bytes 2 & 0	$rd.H[x] = SE16(rs1.B[y]);$  RV32: $(x,y) = (1,2),(0,0)$ RV64: $(x,y) = (3,6),(2,4), (1,2),(0,0)$
3	SUNPKD830 rd, rs1	Signed Unpacking Bytes 3 & 0	$rd.H[x] = SE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,0)$ RV64: $(x,y) = (3,7),(2,4), (1,3),(0,0)$
4	SUNPKD831 rd, rs1	Signed Unpacking Bytes 3 & 1	$rd.H[x] = SE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,1)$ RV64: $(x,y) = (3,7),(2,5), (1,3),(0,1)$
5	SUNPKD832 rd, rs1	Signed Unpacking Bytes 3 & 2	$rd.H[x] = SE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,2)$ RV64: $(x,y) = (3,7),(2,6), (1,3),(0,2)$

No.	Mnemonic	Instruction	Operation
6	ZUNPKD810 rd, rs1	Unsigned Unpacking Bytes 1 & 0	$rd.H[x] = ZE16(rs1.B[y]);$  RV32: $(x,y) = (1,1),(0,0)$ RV64: $(x,y) = (3,5),(2,4), (1,1),(0,0)$
7	ZUNPKD820 rd, rs1	Unsigned Unpacking Bytes 2 & 0	$rd.H[x] = ZE16(rs1.B[y]);$  RV32: $(x,y) = (1,2),(0,0)$ RV64: $(x,y) = (3,6),(2,4), (1,2),(0,0)$
8	ZUNPKD830 rd, rs1	Unsigned Unpacking Bytes 3 & 0	$rd.H[x] = ZE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,0)$ RV64: $(x,y) = (3,7),(2,4), (1,3),(0,0)$
9	ZUNPKD831 rd, rs1	Unsigned Unpacking Bytes 3 & 1	$rd.H[x] = ZE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,1)$ RV64: $(x,y) = (3,7),(2,5), (1,3),(0,1)$
10	ZUNPKD832 rd, rs1	Unsigned Unpacking Bytes 3 & 2	$rd.H[x] = ZE16(rs1.B[y]);$  RV32: $(x,y) = (1,3),(0,2)$ RV64: $(x,y) = (3,7),(2,6), (1,3),(0,2)$

## 3.2. Partial-SIMD Data Processing Instructions

### 3.2.1. 16-bit Packing Instructions

There are 4 instructions here.

*Table 12. 16-bit Packing Instructions*

No.	Mnemonic	Instruction	Operation
1	PKBB16 rd, rs1, rs2	Pack two 16-bit data from Bottoms	$ah0[x] = rs1.W[x].H[0];$ $bh0[x] = rs2.W[x].H[0];$ $rd.W[x] = \text{CONCAT}(ah0[x], bh0[x]);$  (RV32: x=0, RV64: x=1..0)
2	PKBT16 rd, rs1, rs2	Pack two 16-bit data Bottom & Top	$ah0[x] = rs1.W[x].H[0];$ $bh1[x] = rs2.W[x].H[1];$ $rd.W[x] = \text{CONCAT}(ah0[x], bh1[x]);$  (RV32: x=0, RV64: x=1..0)
3	PKTB16 rd, rs1, rs2	Pack two 16-bit data Top & Bottom	$ah1[x] = rs1.W[x].H[1];$ $bh0[x] = rs2.W[x].H[0];$ $rd.W[x] = \text{CONCAT}(ah1[x], bh0[x]);$  (RV32: x=0, RV64: x=1..0)
4	PKTT16 rd, rs1, rs2	Pack two 16-bit data from Tops	$ah1[x] = rs1.W[x].H[1];$ $bh1[x] = rs2.W[x].H[1];$ $rd.W[x] = \text{CONCAT}(ah1[x], bh1[x]);$  (RV32: x=0, RV64: x=1..0)

### 3.2.2. 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	<b>PMULH.W (SMMUL)</b> rd, rs1, rs2	MSW “32 x 32” Signed Multiplication (MSW 32 = 32x32)	$t64[x] = rs1.W[x] s^* rs2.W[x];$ $rd.W[x] = t64[x].W[1];$  (RV32: x=0, RV64: x=1..0)
2	<b>MULHR/PMULHR.W</b> (SMMUL.u) rd, rs1, rs2	MSW “32 x 32” Signed Multiplication with Rounding (MSW 32 = 32x32)	$t64[x] = rs1.W[x] s^* rs2.W[x];$ $rd.W[x] = \text{ROUND}(t64[x]).W[1];$  (RV32: x=0, RV64: x=1..0)
3	<b>MHACC/PMHACC</b> (KMMAC) rd, rs1, rs2	MSW “32 x 32” Signed Multiplication and Saturating Addition (MSW 32 = 32 + 32x32)	$t64[x] = rs1.W[x] s^* rs2.W[x];$ $res[x] = rd.W[x] + t64[x].W[1];$ $rd.W[x] = res[x];$  (RV32: x=0, RV64: x=1..0)
4	<b>MHRACC/PMHRACC</b> .W (KMMAC.u) rd, rs1, rs2	MSW “32 x 32” Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x32)	$t64[x] = rs1.W[x] s^* rs2.W[x];$ $t32[x] = \text{ROUND}(t64[x]).W[1];$ $res[x] = rd.W[x] + t32[x];$ $rd.W[x] = \text{SAT.Q31}(res[x]);$  (RV32: x=0, RV64: x=1..0)
5	KMMSB rd, rs1, rs2	MSW “32 x 32” Signed Multiplication and Saturating Subtraction (MSW 32 = 32 - 32x32)	$t64[x] = rs1.W[x] s^* rs2.W[x];$ $res[x] = rd.W[x] - t64[x].W[1];$ $rd.W[x] = \text{SAT.Q31}(res[x]);$  (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
6	KMMSB.u rd, rs1, rs2	MSW "32 x 32" Signed Multiplication and Saturating Subtraction with Rounding (MSW 32 = 32 - 32x32)	$t64[x] = rs1.W[x] s* rs2.W[x];$ $t32[x] = \text{ROUND}(t64[x]).W[1];$ $res[x] = rd.W[x] - t32[x];$ $rd.W[x] = \text{SAT.Q31}(res[x]);$  (RV32: x=0, RV64: x=1..0)
7	<b>MULQ/PMULQ.W</b> <b>(KWMMUL) rd, rs1,</b> <b>rs2</b>	MSW "32 x 32" Signed Multiplication & Double (MSW 32 = 32x32 << 1)	$t64[x] = rs1.W[x] s* rs2.W[x];$ $s64[x] = \text{SAT.Q63}(t64[x] << 1);$ $rd.W[x] = s64[x].W[1];$  (RV32: x=0, RV64: x=1..0)
8	<b>MULQR/PMULQR.W</b> <b>(KWMMUL.u) rd, rs1,</b> <b>rs2</b>	MSW "32 x 32" Signed Multiplication & Double with Rounding (MSW 32 = 32x32 << 1)	$t64[x] = rs1.W[x] s* rs2.W[x];$ $r65[x] = \text{ROUND}(t64[x] << 1);$ $s64[x] = \text{SAT.Q63}(r65[x]);$ $rd.W[x] = s64[x].W[1];$  (RV32: x=0, RV64: x=1..0)

### 3.2.3. 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	<b>PMULH.W.HE</b> (SMMWB) rd, rs1, rs2	MSW “32 x Bottom 16” Signed Multiplication (MSW 32 = 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[0]);$ $rd.W[x] = mul48[x][47:16];$  (RV32: x=0, RV64: x=1..0)
2	SMMWB.u rd, rs1, rs2	MSW “32 x Bottom 16” Signed Multiplication with Rounding (MSW 32 = 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[0]);$ $rd.W[x] = \text{ROUND}(mul48[x])[47:16];$  (RV32: x=0, RV64: x=1..0)
3	<b>PMULH.W.HO</b> (SMMWT) rd, rs1, rs2	MSW “32 x Top 16” Signed Multiplication (MSW 32 = 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[1]);$ $rd.W[x] = mul48[x][47:16];$  (RV32: x=0, RV64: x=1..0)
4	SMMWT.u rd, rs1, rs2	MSW “32 x Top 16” Signed Multiplication with Rounding (MSW 32 = 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[1]);$ $rd.W[x] = \text{ROUND}(mul48[x])[47:16];$  (RV32: x=0, RV64: x=1..0)
5	<b>PMHACC.W.HE</b> rd, rs1, rs2	MSW “32 x Bottom 16” Signed Multiplication and Addition (MSW 32 = 32 + 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[0]);$ $t[x] = mul48[x][47:16];$ $rd.W[x] = rd.W[x] + t[x];$  (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
6	KMMAWB.u rd, rs1, rs2	MSW "32 x Bottom 16" Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[0]);$ $t[x] = \text{ROUND}(mul48[x])[47:16];$ $rd.W[x] = \text{SAT.Q31}(rd.W[x] + t[x]);$ (RV32: x=0, RV64: x=1..0)
7	<b>PMHACC.W.HO</b> <b>(KMMAWT) rd, rs1, rs2</b>	MSW "32 x Top 16" Signed Multiplication and Addition (MSW 32 = 32 + 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[1]); t[x] = mul48[x][47:16];$ $rd.W[x] = rd.W[x] + t[x];$ (RV32: x=0, RV64: x=1..0)
8	KMMAWT.u rd, rs1, rs2	MSW "32 x Top 16" Signed Multiplication and Saturating Addition with Rounding (MSW 32 = 32 + 32x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul48[x] = a[x] \text{ s}^* (b[x].H[1]);$ $t[x] = \text{ROUND}(mul48[x])[47:16];$ $rd.W[x] = \text{SAT.Q31}(rd.W[x] + t[x]);$ (RV32: x=0, RV64: x=1..0)
9	KMMWB2 rd, rs1, rs2	MSW "32 x Bottom 16" Saturating Signed Multiplication and double (MSW 32 = (32x16) << 1)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $\text{if } ((a[x] == 0x80000000) \& \& (b[x].H[0] == 0x8000)) \{$ $t[x] = 0x7fffffff; OV = 1;$ $\} \text{ else } \{$ $mul48[x] = a[x] \text{ s}^* (b[x].H[0]);$ $t[x] = (mul48[x] << 1)[47:16];$ $\}$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
10	KMMWB2.u rd, rs1, rs2	MSW "32 x Bottom 16" Saturating Signed Multiplication and double with Rounding (MSW 32 = (32x16) << 1)	<pre>a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[0]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[0]);     t[x] = ROUND(mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = t[x];  (RV32: x=0, RV64: x=1..0)</pre>
11	KMMWT2 rd, rs1, rs2	MSW "32 x Top 16" Saturating Signed Multiplication and double (MSW 32 = (32x16) << 1)	<pre>a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[1]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[1]);     t[x] = (mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = t[x];  (RV32: x=0, RV64: x=1..0)</pre>
12	KMMWT2.u rd, rs1, rs2	MSW "32 x Top 16" Saturating Signed Multiplication and double with Rounding (MSW 32 = (32x16) << 1)	<pre>a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[1]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[1]);     t[x] = ROUND(mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = t[x];  (RV32: x=0, RV64: x=1..0)</pre>

No.	Mnemonic	Instruction	Operation
13	KMMAWB2 rd, rs1, rs2	MSW "32 x Bottom 16" Signed Multiplication & double and Saturating Addition (MSW 32 = 32 + (32x16)<<1)	<pre> a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[0]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[0]);     t[x] = (mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = SAT.Q31(rd.W[x] + t[x]);  (RV32: x=0, RV64: x=1..0) </pre>
14	KMMAWB2.u rd, rs1, rs2	MSW "32 x Bottom 16" Signed Multiplication & double and Saturating Addition with Rounding (MSW 32 = 32 + (32x16)<<1)	<pre> a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[0]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[0]);     t[x] = ROUND(mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = SAT.Q31(rd.W[x] + t[x]);  (RV32: x=0, RV64: x=1..0) </pre>
15	KMMAWT2 rd, rs1, rs2	MSW "32 x Top 16" Signed Multiplication & double and Saturating Addition (MSW 32 = 32 + (32x16)<<1)	<pre> a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[1]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[1]);     t[x] = (mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = SAT.Q31(rd.W[x] + t[x]);  (RV32: x=0, RV64: x=1..0) </pre>

No.	Mnemonic	Instruction	Operation
16	KMMAWT2.u rd, rs1, rs2	MSW "32 x Top 16" Signed Multiplication & double and Saturating Addition with Rounding (MSW 32 = 32 + $(32 \times 16) \ll 1$ )	<pre> a[x]=rs1.W[x]; b[x]=rs2.W[x]; if ((a[x]==0x80000000) &amp;&amp;     (b[x].H[1]==0x8000)) {     t[x] = 0x7fffffff; OV=1; } else {     mul48[x] = a[x] s* (b[x].H[1]);     t[x] = ROUND(mul48[x]&lt;&lt;1)[47:16]; } rd.W[x] = SAT.Q31(rd.W[x] + t[x]);  (RV32: x=0, RV64: x=1..0) </pre>

### 3.2.4. 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	<b>PMUL.W.HEE (SMBB16)</b> rd, rs1, rs2	Signed Multiply Even 16 & Even 16 (32 = 16x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $rd.W[x] = a[x].H[0] \text{ s}^* b[x].H[0];$ (RV32: x=0, RV64: x=1..0)
2	<b>MULW.HEO (SMBT16)</b> rd, rs1, rs2	Signed Multiply Even 16 & Odd 16 (32 = 16x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $rd.W[x] = a[x].H[0] \text{ s}^* b[x].H[1];$ (RV32: x=0, RV64: x=1..0)
3	<b>PMUL.W.HOO (SMTT16)</b> rd, rs1, rs2	Signed Multiply Odd 16 & Odd 16 (32 = 16x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $rd.W[x] = a[x].H[1] \text{ s}^* b[x].H[1];$ (RV32: x=0, RV64: x=1..0)
4	KMDA rd, rs1, rs2	Two "16x16" and Signed Addition (32 = 16x16 + 16x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $\text{mul1}[x] = a[x].H[1] \text{ s}^* b[x].H[1];$ $\text{mul2}[x] = a[x].H[0] \text{ s}^* b[x].H[0];$ $t[x] = \text{SAT.Q31}(\text{mul1}[x] + \text{mul2}[x]);$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)
5	KMXDA rd, rs1, rs2	Two Crossed "16x16" and Signed Addition (32 = 16x16 + 16x16)	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $\text{mul1}[x] = a[x].H[1] \text{ s}^* b[x].H[0];$ $\text{mul2}[x] = a[x].H[0] \text{ s}^* b[x].H[1];$ $t[x] = \text{SAT.Q31}(\text{mul1}[x] + \text{mul2}[x]);$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
6	SMDS rd, rs1, rs2	Two "16x16" and Signed Subtraction ( $32 = 16x16 - 16x16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^*$ $b[x].H[0]; t[x] = mul1[x] - mul2[x];$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)
7	SMDRS rd, rs1, rs2	Two "16x16" and Signed Reversed Subtraction ( $32 = 16x16 - 16x16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^*$ $b[x].H[0]; t[x] = mul2[x] - mul1[x];$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)
8	SMXDS rd, rs1, rs2	Two Crossed "16x16" and Signed Subtraction ( $32 = 16x16 - 16x16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[0];$ $mul2[x] = a[x].H[0] s^*$ $b[x].H[1]; t[x] = mul1[x] - mul2[x];$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)
9	PMACC.W.HEE (KMABB) rd, rs1, rs2	"Even 16 x Even 16" with 32-bit Signed Addition ( $32 = 32 + 16x16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul[x] = a[x].H[0] s^* b[x].H[0];$ $t[x] = rd.W[x] + mul[x];$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)
10	PMAC.W.HEO (KMABT) rd, rs1, rs2	"Even 16 x Odd 16" with 32-bit Signed Addition ( $32 = 32 + 16x16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul[x] = a[x].H[0] s^* b[x].H[1];$ $t[x] = rd.W[x] + mul[x];$ $rd.W[x] = t[x];$ (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
11	PMAC.W.HOO (KMATT) rd, rs1, rs2	"Odd 16 x Odd 16" with 32-bit Signed Addition ( $32 = 32 + 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul[x] = a[x].H[1] s^*$ $b[x].H[1]; t[x] = rd.W[x] +$ $mul[x];$ $rd.W[x] = t[x];$  (RV32: x=0, RV64: x=1..0)
12	KMADA rd, rs1, rs2	Two "16x16" with 32-bit Signed Double Addition ( $32 = 32 + 16 \times 16 +$ $16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^* b[x].H[0];$ $t[x] = rd.W[x] + mul1[x] + mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)
13	KMAXDA rd, rs1, rs2	Two Crossed "16x16" with 32- bit Signed Double Addition ( $32 = 32 + 16 \times 16 +$ $16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[0];$ $mul2[x] = a[x].H[0] s^* b[x].H[1];$ $t[x] = rd.W[x] + mul1[x] + mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)
14	KMADS rd, rs1, rs2	Two "16x16" with 32-bit Signed Addition and Subtraction ( $32 = 32 + 16 \times 16$ $- 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^* b[x].H[0];$ $t[x] = rd.W[x] + mul1[x] - mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)
15	KMADRS rd, rs1, rs2	Two "16x16" with 32-bit Signed Addition and Reversed Subtraction ( $32 = 32 + 16 \times 16$ $- 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^* b[x].H[0];$ $t[x] = rd.W[x] + mul2[x] - mul1[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
16	KMAXDS rd, rs1, rs2	Two Crossed "16x16" with 32-bit Signed Addition and Subtraction ( $32 = 32 + 16 \times 16 - 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[0];$ $mul2[x] = a[x].H[0] s^* b[x].H[1];$ $t[x] = rd.W[x] + mul1[x] - mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)
17	KMSDA rd, rs1, rs2	Two "16x16" with 32-bit Signed Double Subtraction ( $32 = 32 - 16 \times 16 - 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[1];$ $mul2[x] = a[x].H[0] s^* b[x].H[0];$ $t[x] = rd.W[x] - mul1[x] - mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)
18	KMSXDA rd, rs1, rs2	Two Crossed "16x16" with 32-bit Signed Double Subtraction ( $32 = 32 - 16 \times 16 - 16 \times 16$ )	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $mul1[x] = a[x].H[1] s^* b[x].H[0];$ $mul2[x] = a[x].H[0] s^* b[x].H[1];$ $t[x] = rd.W[x] - mul1[x] - mul2[x];$ $rd.W[x] = SAT.Q31(t[x]);$  (RV32: x=0, RV64: x=1..0)

### 3.2.5. 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 rd, rs1, rs2	"16 x 16" with 64-bit Signed Addition ( $64 = 64 + 16 \times 16$ )	<p>RV32:</p> <pre>a64 = r[rs1U].r[rs1L]; mul = rs2.H[1] s* rs2.H[0]; t64 = a64 + mul; r[dU].r[dL] = t64;</pre> <p>RV64:</p> <pre>a64 = rs1; tw = rs2.W[1]; bw = rs2.W[0]; mul1 = tw.H[1] s* tw.H[0]; mul2 = bw.H[1] s* bw.H[0]; rd = a64 + mul1 + mul2;</pre>

### 3.2.6. Miscellaneous Instructions

There are 7 instructions here.

Table 17. Partial-SIMD Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	SCLIP32 rd, rs1, imm5u	Signed Clip Value	$n = \text{imm5u};$ $rd = \text{SAT.Qn}(\text{rs1.W}[x]);$ (RV32: x=0, RV64: x=1..0)
2	UCLIP32 rd, rs1, imm5u	Unsigned Clip Value	$m = \text{imm5u};$ $rd = \text{SAT.Um}(\text{rs1.W}[x]);$ (RV32: x=0, RV64: x=1..0)
3	CLRS32 rd, rs1	32-bit Count Leading Redundant Sign	$rd.W[x] = \text{CLRS}(\text{rs1.W}[x])$ (RV32: x=0, RV64: x=1..0)
4	CLZ32 rd, rs1	32-bit Count Leading Zero	$rd.W[x] = \text{CLZ}(\text{rs1.W}[x])$ (RV32: x=0, RV64: x=1..0)
5	PDIFSUMU.B (PBSAD) rd, rs1, rs2	Parallel Byte Sum of Absolute Difference	$d[x] = \text{ABS}(\text{rs1.B}[x] - \text{rs2.B}[x]);$ $rd = \text{SUM}(d[x]);$ (RV32: x=3..0, RV64: x=7..0)
6	PBSADA rd, rs1, rs2	Parallel Byte Sum of Absolute Difference Accumulation	$d[x] = \text{ABS}(\text{rs1.B}[x] - \text{rs2.B}[x]);$ $rd = rd + \text{SUM}(d[x]);$ (RV32: x=3..0, RV64: x=7..0)

### 3.2.7. 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 rd, rs1, rs2	Four signed "8x8" with 32-bit Signed Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $m0[x] = a[x].B[0] s^* b[x].B[0];$ $m1[x] = a[x].B[1] s^* b[x].B[1];$ $m2[x] = a[x].B[2] s^* b[x].B[2];$ $m3[x] = a[x].B[3] s^* b[x].B[3];$ $rd.W[x] = rd.W[x] + m3[x] + m2[x] + m1[x] + m0[x];$ $(RV32: x=0, RV64: x=1..0)$
2	UMAQA rd, rs1, rs2	Four unsigned "8x8" with 32-bit Unsigned Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $m0[x] = a[x].B[0] u^* b[x].B[0];$ $m1[x] = a[x].B[1] u^* b[x].B[1];$ $m2[x] = a[x].B[2] u^* b[x].B[2];$ $m3[x] = a[x].B[3] u^* b[x].B[3];$ $rd.W[x] = rd.W[x] + m3[x] + m2[x] + m1[x] + m0[x];$ $(RV32: x=0, RV64: x=1..0)$
3	SMAQA.SU rd, rs1, rs2	Four "signed 8 x unsigned 8" with 32-bit Signed Addition $(32 = 32 + 8x8 + 8x8 + 8x8 + 8x8)$	$a[x] = rs1.W[x]; b[x] = rs2.W[x];$ $m0[x] = a[x].B[0] su^* b[x].B[0];$ $m1[x] = a[x].B[1] su^* b[x].B[1];$ $m2[x] = a[x].B[2] su^* b[x].B[2];$ $m3[x] = a[x].B[3] su^* b[x].B[3];$ $rd.W[x] = rd.W[x] + m3[x] + m2[x] + m1[x] + m0[x];$ $(RV32: x=0, RV64: x=1..0)$

### 3.3. 64-bit Data Computation Instructions

#### 3.3.1. 64-bit Addition & Subtraction Instructions

Table 19. 64-bit Add/Subtract Instructions

No.	Mnemonic	Instruction	Operation
1	ADDD (ADD64) rd, rs1, rs2	64-bit Addition	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = a64 + b64;$ $r[dU].r[dL] = t64;$ <p>(RV32 Only)</p>
2	RADD64 rd, rs1, rs2	64-bit Signed Averaging Addition	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = (a64 + b64) s\gg 1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = (rs1 + rs2) s\gg 1;$
3	URADD64 rd, rs1, rs2	64-bit Unsigned Averaging Addition	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $a65 = \text{CONCAT}(1'b0, a64);$ $b65 = \text{CONCAT}(1'b0, b64);$ $t64 = (a65 + b65) u\gg 1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $a65 = \text{CONCAT}(1'b0, rs1);$ $b65 = \text{CONCAT}(1'b0, rs2);$ $rd = (a65 + b65) u\gg 1;$

No.	Mnemonic	Instruction	Operation
4	KADD64 rd, rs1, rs2	64-bit Signed Saturating Addition	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = \text{SAT.Q63}(a64 + b64);$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = \text{SAT.Q63}(rs1 + rs2);$
5	UKADD64 rd, rs1, rs2	64-bit Unsigned Saturating Addition	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = \text{SAT.U64}(a64 + b64);$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = \text{SAT.U64}(rs1 + rs2);$
6	<b>SUBD (SUB64) rd, rs1, rs2</b>	64-bit Subtraction	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = a64 - b64;$ $r[dU].r[dL] = t64;$ <p>(RV32 Only)</p>
7	RSUB64 rd, rs1, rs2	64-bit Signed Averaging Subtraction	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = (a64 - b64) \text{ s} >> 1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = (rs1 - rs2) \text{ s} >> 1;$

No.	Mnemonic	Instruction	Operation
8	URSUB64 rd, rs1, rs2	64-bit Unsigned Averaging Subtraction	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $a65 = \text{CONCAT}(1'b0, a64);$ $b65 = \text{CONCAT}(1'b0, b64);$ $t64 = (a65 - b65) \text{ u}>> 1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $a65 = \text{CONCAT}(1'b0, rs1);$ $b65 = \text{CONCAT}(1'b0, rs2);$ $rd = (a65 - b65) \text{ u}>> 1;$
9	KSUB64 rd, rs1, rs2	64-bit Signed Saturating Subtraction	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = \text{SAT.Q63}(a64 - b64);$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = \text{SAT.Q63}(rs1 - rs2);$
10	UKSUB64 rd, rs1, rs2	64-bit Unsigned Saturating Subtraction	<p>RV32:</p> $a64 = r[rs1U].r[rs1L];$ $b64 = r[rs2U].r[rs2L];$ $t64 = \text{SAT.U64}(a64 - b64);$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $rd = \text{SAT.U64}(rs1 - rs2);$

### 3.3.2. 32-bit Multiply with 64-bit Add/Subtract Instructions

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

No.	Mnemonic	Instruction	Operation
1	PM2ADDA.W (SMAR64) rd, rs1, rs2	32x32 with 64-bit Signed Addition	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 + rs1 \ s^* rs2;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \ s^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \ s^* rs2.W[1];</math>  <math>rd = rd + m0 + m1;</math></p>
2	SMSR64 rd, rs1, rs2	32x32 with 64-bit Signed Subtraction	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 - rs1 \ s^* rs2;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \ s^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \ s^* rs2.W[1];</math>  <math>rd = rd - m0 - m1;</math></p>
3	UMAR64 rd, rs1, rs2	32x32 with 64-bit Unsigned Addition	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 + rs1 \ u^* rs2;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \ u^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \ u^* rs2.W[1];</math>  <math>rd = rd + m0 + m1;</math></p>

No.	Mnemonic	Instruction	Operation
4	UMSR64 rd, rs1, rs2	32x32 with 64-bit Unsigned Subtraction	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 - rs1 \cdot u^* rs2;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \cdot u^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \cdot u^* rs2.W[1];</math>  <math>rd = rd - m0 - m1;</math></p>
5	KMAR64 rd, rs1, rs2	32x32 with Saturating 64-bit Signed Addition	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = SAT.Q63(c64 + rs1 \cdot s^* rs2);</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \cdot s^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \cdot s^* rs2.W[1];</math>  <math>rd = SAT.Q63(rd + m0 + m1);</math></p>
6	KMSR64 rd, rs1, rs2	32x32 with Saturating 64-bit Signed Subtraction	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = SAT.Q63(c64 - rs1 \cdot s^* rs2);</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \cdot s^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \cdot s^* rs2.W[1];</math>  <math>rd = SAT.Q63(rd - m0 - m1);</math></p>
7	UKMAR64 rd, rs1, rs2	32x32 with Saturating 64-bit Unsigned Addition	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = SAT.U64(c64 + rs1 \cdot u^* rs2);</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0] \cdot u^* rs2.W[0];</math>  <math>m1 = rs1.W[1] \cdot u^* rs2.W[1];</math>  <math>rd = SAT.U64(rd + m0 + m1);</math></p>

No.	Mnemonic	Instruction	Operation
8	UKMSR64 rd, rs1, rs2	32x32 with Saturating 64-bit Unsigned Subtraction	<p>RV32:</p> $c64 = r[dU].r[dL];$ $t64 = \text{SAT.U64}(c64 - rs1.u^* rs2);$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $m0 = rs1.W[0] u^* rs2.W[0];$ $m1 = rs1.W[1] u^* rs2.W[1];$ $rd = \text{SAT.U64}(rd - m0 - m1);$

### 3.3.3. 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 rd, rs1, rs2	"Bottom 16 x Bottom 16" with 64-bit Signed Addition (64 = 64 + 16x16)	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 + rs1.L \ s^* rs2.L;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0].H[0] \ s^*</math>  <math>rs2.W[0].H[0];</math>  <math>m1 = rs1.W[1].H[0] \ s^*</math>  <math>rs2.W[1].H[0];</math>  <math>rd = rd + m0 + m1;</math></p>
2	SMALBT rd, rs1, rs2	"Bottom 16 x Top 16" with 64-bit Signed Addition (64 = 64 + 16x16)	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 + rs1.L \ s^* rs2.H;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0].H[0] \ s^*</math>  <math>rs2.W[0].H[1];</math>  <math>m1 = rs1.W[1].H[0] \ s^*</math>  <math>rs2.W[1].H[1];</math>  <math>rd = rd + m0 + m1;</math></p>
3	SMALTT rd, rs1, rs2	"Top 16 x Top 16" with 64-bit Signed Addition (64 = 64 + 16x16)	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>t64 = c64 + rs1.H \ s^* rs2.H;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0].H[1] \ s^*</math>  <math>rs2.W[0].H[1];</math>  <math>m1 = rs1.W[1].H[1] \ s^*</math>  <math>rs2.W[1].H[1];</math>  <math>rd = rd + m0 + m1;</math></p>

No.	Mnemonic	Instruction	Operation
4	<b>PM4ADDA.H (SMALDA)</b> rd, rs1, rs2	Two "16x16" with 64-bit Signed Double Addition $(64 = 64 + 16 \times 16 + 16 \times 16)$	<p>RV32:</p> $c64 = r[dU].r[dL];$ $m0 = rs1.H \text{ s}^* rs2.H;$ $m1 = rs1.L \text{ s}^* rs2.L;$ $t64 = c64 + m0 + m1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $m0 = rs1.W[0].H[0] \text{ s}^*$ $rs1.W[0].H[0];$ $m1 = rs1.W[0].H[1] \text{ s}^*$ $rs1.W[0].H[1];$ $m2 = rs1.W[1].H[0] \text{ s}^*$ $rs1.W[1].H[0];$ $m3 = rs1.W[1].H[1] \text{ s}^*$ $rs1.W[1].H[1];$ $rd = rd + \text{SUM}(m0 \sim 3);$
5	SMALXDA rd, rs1, rs2	Two Crossed "16x16" with 64-bit Signed Double Addition $(64 = 64 + 16 \times 16 + 16 \times 16)$	<p>RV32:</p> $c64 = r[dU].r[dL];$ $m0 = rs1.H \text{ s}^* rs2.L;$ $m1 = rs1.L \text{ s}^* rs2.H;$ $t64 = c64 + m0 + m1;$ $r[dU].r[dL] = t64;$ <p>RV64:</p> $m0 = rs1.W[0].H[0] \text{ s}^*$ $rs1.W[0].H[1];$ $m1 = rs1.W[0].H[1] \text{ s}^*$ $rs1.W[0].H[0];$ $m2 = rs1.W[1].H[0] \text{ s}^*$ $rs1.W[1].H[1];$ $m3 = rs1.W[1].H[1] \text{ s}^*$ $rs1.W[1].H[0];$ $rd = rd + \text{SUM}(m0 \sim 3);$

No.	Mnemonic	Instruction	Operation
6	SMALDS rd, rs1, rs2	<p>Two "16x16" with 64-bit Signed Addition and Subtraction  <math>(64 = 64 + 16 \times 16 - 16 \times 16)</math></p>	<p>c64 = r[dU].r[dL];  m0 = rs1.H s* rs2.H;  m1 = rs1.L s* rs2.L;  t64 = c64 + m0 - m1;  r[dU].r[dL] = t64;</p> <p>RV64:  m0 = rs1.W[0].H[1] s* rs2.W[0].H[1];  m1 = rs1.W[0].H[0] s* rs2.W[0].H[0];  m2 = rs1.W[1].H[1] s* rs2.W[1].H[1];  m3 = rs1.W[1].H[0] s* rs2.W[1].H[0];  s0 = m0 - m1;  s1 = m2 - m3;  rd = rd + s0 + s1;</p>
7	SMALDRS rd, rs1, rs2	<p>Two "16x16" with 64-bit Signed Addition and Reversed Subtraction  <math>(64 = 64 + 16 \times 16 - 16 \times 16)</math></p>	<p>RV32:  c64 = r[dU].r[dL];  m0 = rs1.L s* rs2.L;  m1 = rs1.H s* rs2.H;  t64 = c64 + m0 - m1;  r[dU].r[dL] = t64;</p> <p>RV64:  m0 = rs1.W[0].H[0] s* rs2.W[0].H[0];  m1 = rs1.W[0].H[1] s* rs2.W[0].H[1];  m2 = rs1.W[1].H[0] s* rs2.W[1].H[0];  m3 = rs1.W[1].H[1] s* rs2.W[1].H[1];  s0 = m0 - m1;  s1 = m2 - m3;  rd = rd + s0 + s1;</p>

No.	Mnemonic	Instruction	Operation
8	SMALXDS rd, rs1, rs2	Two Crossed "16x16" with 64-bit Signed Addition and Subtraction $(64 = 64 + 16 \times 16 - 16 \times 16)$	<p>RV32:</p> $\begin{aligned} c64 &= r[dU].r[dL]; \\ m0 &= rs1.H \text{ s* } rs2.L; \\ m1 &= rs1.L \text{ s* } rs2.H; \\ t64 &= c64 + m0 - m1; \\ r[dU].r[dL] &= t64; \end{aligned}$ <p>RV64:</p> $\begin{aligned} m0 &= rs1.W[0].H[1] \text{ s* } \\ &rs2.W[0].H[0]; \\ m1 &= rs1.W[0].H[0] \text{ s* } \\ &rs2.W[0].H[1]; \\ m2 &= rs1.W[1].H[1] \text{ s* } \\ &rs2.W[1].H[0]; \\ m3 &= rs1.W[1].H[0] \text{ s* } \\ &rs2.W[1].H[1]; \\ s0 &= m0 - m1; \\ s1 &= m2 - m3; \\ rd &= rd + s0 + s1; \end{aligned}$
9	SMSLDA rd, rs1, rs2	Two "16x16" with 64-bit Signed Double Subtraction $(64 = 64 - 16 \times 16 - 16 \times 16)$	<p>RV32:</p> $\begin{aligned} c64 &= r[dU].r[dL]; \\ m0 &= rs1.H \text{ s* } rs2.H; \\ m1 &= rs1.L \text{ s* } rs2.L; \\ t64 &= c64 - m0 - m1; \\ r[dU].r[dL] &= t64; \end{aligned}$ <p>RV64:</p> $\begin{aligned} m0 &= rs1.W[0].H[0] \text{ s* } \\ &rs2.W[0].H[0]; \\ m1 &= rs1.W[0].H[1] \text{ s* } \\ &rs2.W[0].H[1]; \\ m2 &= rs1.W[1].H[0] \text{ s* } \\ &rs2.W[1].H[0]; \\ m3 &= rs1.W[1].H[1] \text{ s* } \\ &rs2.W[1].H[1]; \\ s0 &= - m0 - m1; \\ s1 &= - m2 - m3; \\ rd &= rd + s0 + s1; \end{aligned}$

No.	Mnemonic	Instruction	Operation
10	SMSLXDA rd, rs1, rs2	<p>Two Crossed "16x16" with 64-bit Signed Double Subtraction (<math>64 = 64 - 16 \times 16 - 16 \times 16</math>)</p>	<p>RV32:  <math>c64 = r[dU].r[dL];</math>  <math>m0 = rs1.H \text{ s}^* rs2.L;</math>  <math>m1 = rs1.L \text{ s}^* rs2.H;</math>  <math>t64 = c64 - m0 - m1;</math>  <math>r[dU].r[dL] = t64;</math></p> <p>RV64:  <math>m0 = rs1.W[0].H[0] \text{ s}^*</math>  <math>rs2.W[0].H[1];</math>  <math>m1 = rs1.W[0].H[1] \text{ s}^*</math>  <math>rs2.W[0].H[0];</math>  <math>m2 = rs1.W[1].H[0] \text{ s}^*</math>  <math>rs2.W[1].H[1];</math>  <math>m3 = rs1.W[1].H[1] \text{ s}^*</math>  <math>rs2.W[1].H[0];</math>  <math>s0 = - m0 - m1;</math>  <math>s1 = - m2 - m3;</math>  <math>rd = rd + s0 + s1;</math></p>

## 3.4. Non-SIMD Instructions

### 3.4.1. 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 rd, rs1, rs2	Add with Q15 saturation	$a17 = SE17(rs1.H[0]);$ $b17 = SE17(rs2.H[0]);$ $t17 = a17 + b17;$ $k16 = SAT.Q15(t17);$ $rd = SE\_XLEN(k16);$
2	KSUBH rd, rs1, rs2	Subtract with Q15 saturation	$a17 = SE17(rs1.H[0]);$ $b17 = SE17(rs2.H[0]);$ $t17 = a17 - b17;$ $k16 = SAT.Q15(t17);$ $rd = SE\_XLEN(k16);$
3	KHMBB rd, rs1, rs2	Multiply the first 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q15 number.	$a0 = rs1.H[0];$ $b0 = rs2.H[0];$ $rd = SAT.Q15((a0 * b0) s>> 15);$
4	KHMBT rd, rs1, rs2	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.	$a0 = rs1.H[0];$ $b1 = rs2.H[1];$ $rd = SAT.Q15((a0 * b1) s>> 15);$

No.	Mnemonic	Instruction	Operation
5	KHMTT rd, rs1, rs2	Multiply the second 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q15 number.	<pre>a1 = rs1.H[1]; b1 = rs2.H[1]; rd = SAT.Q15((a1 * b1) s&gt;&gt; 15);</pre>
6	UKADDH rd, rs1, rs2	Add with I16 saturation	<pre>a17 = ZE17(rs1.H[0]); b17 = ZE17(rs2.H[0]); t17 = a17 + b17; uk16 = SAT.U16(t17); rd = SE_XLEN(uk16);</pre>
7	UKSUBH rd, rs1, rs2	Subtract with I16 saturation	<pre>a17 = ZE17(rs1.H[0]); b17 = ZE17(rs2.H[0]); t17 = a17 - b17; uk16 = SAT.U16(t17); rd = SE_XLEN(uk16);</pre>

### 3.4.2. 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	SADD (KADDW) rd, rs1, rs2	Add with Q31 saturation	<p>RV32:  <math>rd = SAT.Q31(rs1 + rs2);</math></p> <p>RV64:  <math>a0 = rs1.W[0];</math>  <math>b0 = rs2.W[0];</math>  <math>rd = SE\_XLEN(SAT.Q31(a0 + b0));</math></p>
2	SADDU (UKADDW) rd, rs1, rs2	Unsigned Add with U32 saturation	<p>RV32:  <math>a0 = CONCAT(1'b0, rs1);</math>  <math>b0 = CONCAT(1'b0, rs2);</math>  <math>rd = SAT.U32(a0 + b0);</math></p> <p>RV64:  <math>a0 = CONCAT(1'b0, rs1.W[0]);</math>  <math>b0 = CONCAT(1'b0, rs2.W[0]);</math>  <math>rd = ZE(SAT.U32(a0 + b0));</math></p>
3	SSUB (KSUBW) rd, rs1, rs2	Subtract with Q31 saturation	<p>RV32:  <math>rd = SAT.Q31(rs1 - rs2);</math></p> <p>RV64:  <math>a0 = rs1.W[0];</math>  <math>b0 = rs2.W[0];</math>  <math>rd = SE\_XLEN(SAT.Q31(a0 - b0));</math></p>
4	SSUBU (UKSUBW) rd, rs1, rs2	Unsigned Subtract with U32 saturation	<p>RV32:  <math>a0 = CONCAT(1'b0, rs1);</math>  <math>b0 = CONCAT(1'b0, rs2);</math>  <math>rd = SAT.U32(a0 - b0);</math></p> <p>RV64:  <math>a0 = CONCAT(1'b0, rs1.W[0]);</math>  <math>b0 = CONCAT(1'b0, rs2.W[0]);</math>  <math>rd = ZE(SAT.U32(a0 - b0));</math></p>

No.	Mnemonic	Instruction	Operation
5	KDMBB rd, rs1, rs2	Multiply the first 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number.	<pre>a0 = rs1.H[0]; b0 = rs2.H[0]; m0 = (a0 s* b0) &lt;&lt; 1; rd = SAT.Q31(m0);</pre>
6	KDMBT rd, rs1, rs2	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.	<pre>a0 = rs1.H[0]; b1 = rs2.H[1]; m0 = (a0 s* b1) &lt;&lt; 1; rd = SAT.Q31(m0);</pre>
7	KDMTT rd, rs1, rs2	Multiply the second 16-bit Q15 elements of two registers and transform the Q30 result into a saturated Q31 number.	<pre>a1 = rs1.H[1]; b1 = rs2.H[1]; m0 = (a1 s* b1) &lt;&lt; 1; rd = SAT.Q31(m0);</pre>
8	KSLRAW rd, rs1, rs2	Shift Left Logical with Q31 Saturation or Shift Right Arithmetic	<pre>if (rs2[5:0] &gt;=0) {     rd = SAT.Q31(rs1 &lt;&lt; rs2[5:0]); } else {     rd = (rs1 s&gt;&gt; -rs2[5:0]); }</pre>
9	KSLRAW.u rd, rs1, rs2	Shift Left Logical with Q31 Saturation or Rounding Shift Right Arithmetic	<pre>if (rs2[5:0] &gt;=0) {     rd = SAT.Q31(rs1 &lt;&lt; rs2[5:0]); } else {     rd = ROUND(rs1 s&gt;&gt; -rs2[5:0]); }</pre>
10	KSLLW rd, rs1, rs2	Saturating Shift Left Logical for 32-bit Word	<pre>w0 = rs1.W[0]; rd = SE_XLEN(SAT.Q31(w0 &lt;&lt; rs2[4:0]));</pre>

No.	Mnemonic	Instruction	Operation
11	KSLLIW rd, rs1, imm5u	Saturating Shift Left Logical Immediate for 32-bit Word	$w0 = rs1.W[0];$ $rd = SE\_XLEN(SAT.Q31(w0 << imm5u));$
12	KDMABB rd, rs1, rs2	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.	$m0 = (rs1.H[0] * rs2.H[0]) << 1;$ $res = rd.W[0] + SAT.Q31(m0);$ $rd = SE\_XLEN(SAT.Q31(res));$
13	KDMABT rd, rs1, rs2	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.	$m0 = (rs1.H[0] * rs2.H[1]) << 1;$ $res = rd.W[0] + SAT.Q31(m0);$ $rd = SE\_XLEN(SAT.Q31(res));$
14	KDMATT rd, rs1, rs2	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.	$m0 = (rs1.H[1] * rs2.H[1]) << 1;$ $res = rd.W[0] + SAT.Q31(m0);$ $rd = SE\_XLEN(SAT.Q31(res));$
15	<b>ABS (KABSW) rd, rs1</b>	32-bit Absolute Value (scalar version)	RV32: $rd = ABS(rs1);$  RV64: $rd = SE64(ABS(rs1.W[0]));$

### 3.4.3. 32-bit Computation Instructions

There are 7 instructions here.

Table 24. 32-bit Computation Instructions

No.	Mnemonic	Instruction	Operation
1	AADD (RADDW) rd, rs1, rs2	32-bit Signed Averaging Addition	$res = (rs1.W[0] + rs2.W[0]) s\gg 1;$ $rd = SE\_XLEN(res);$
2	AADDU (URADDW) rd, rs1, rs2	32-bit Unsigned Averaging Addition	$a0 = \text{CONCAT}(1'b0, rs1.W[0]);$ $b0 = \text{CONCAT}(1'b0, rs2.W[0]);$ $res = (a0 + b0) u\gg 1;$ $rd = SE\_XLEN(res);$
3	ASUB (RSUBW) rd, rs1, rs2	32-bit Signed Averaging Subtraction	$res = (rs1.W[0] - rs2.W[0]) s\gg 1;$ $rd = SE\_XLEN(res);$
4	ASUBU (URSUBW) rd, rs1, rs2	32-bit Unsigned Averaging Subtraction	$a0 = \text{CONCAT}(1'b0, rs1.W[0]);$ $b0 = \text{CONCAT}(1'b0, rs2.W[0]);$ $res = (a0 - b0) u\gg 1;$ $rd = SE\_XLEN(res);$
5	MULR64 rd, rs1, rs2	Multiply Word Unsigned to 64-bit data	RV32: $mres[63:0] = rs1 u^* rs2;$ $r[dU] = mres.W[1];$ $r[dL] = mres.W[0];$  RV64: $rd = rs1.W[0] u^* rs2.W[0];$
6	MULSR64 rd, rs1, rs2	Multiply Word Signed to 64-bit data	RV32: $mres[63:0] = rs1 s^* rs2;$ $r[dU] = mres.W[1];$ $r[dL] = mres.W[0];$  RV64: $rd = rs1.W[0] s^* rs2.W[0];$
7	MSUBR32 rd, rs1, rs2	Multiply and Subtract from 32-bit Word	RV32: $mres = rs1 * rs2;$ $rd = rd - mres.W[0];$  RV64: $mres = rs1.W[0] * rs2.W[0];$ $tres[31:0] = rd.W[0] - mres.W[0];$ $rd = SE64(tres[31:0]);$

### 3.4.4. Overflow/Saturation 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 rd	Read vxsat.OV to rd.	$rd = \text{ZE}(\text{vxsat.OV});$
2	CLROV	Clear vsat.OV flag	$\text{vxsat.OV} = 0;$

### 3.4.5. Miscellaneous Instructions

There are 13 instructions here.

Table 26. Non-SIMD Miscellaneous Instructions

No.	Mnemonic	Instruction	Operation
1	AVE rd, rs1, rs2	Average with rounding	
2	SRA.u rd, rs1, rs2	Rounding Shift Right Arithmetic	RV32: rd = ROUND(rs1 s>> rs2[4:0]);  RV64: rd = ROUND(rs1 s>> rs2[5:0]);
3	SRAI.u rd, rs1, imm5u/imm6u	Rounding Shift Right Arithmetic Immediate	RV32: rd = ROUND(rs1 s>> imm5u);  RV64: rd = ROUND(rs1 s>> imm6u);
4	BITREV rd, rs1, rs2	Bit Reverse	RV32: msb = rs2[4:0]; rev[0:msb] = rs1[msb:0]; rd = ZE32(rev[msb:0]);  RV64: msb = rs2[5:0]; rev[0:msb] = rs1[msb:0]; rd = ZE64(rev[msb:0]);
5	BITREVI rd, rs1, imm5u/imm6u	Bit Reverse Immediate	RV32: msb = imm5u; rev[0:msb] = rs1[msb:0]; rd = ZE32(rev[msb:0]);  RV64: msb = imm6u; rev[0:msb] = rs1[msb:0]; rd = ZE64(rev[msb:0]);
6	WEXT rd, rs1, rs2	Extract 32-bit from a 64-bit value	RV32: a64 = r[rs1U].r[rs1L]; lsb = rs2[4:0]; exword = a64[(31+lsb):lsb]; rd = SE32(exword);  RV64: a64 = rs1; lsb = rs2[4:0]; exword = a64[(31+lsb):lsb]; rd = SE64(exword);

No.	Mnemonic	Instruction	Operation
7	WEXTI rd, rs1, imm5u	Extract 32-bit from a 64-bit value Immediate	RV32: a64 = r[rs1U].r[rs1L]; lsb = imm5u; exword = a64[(31+lsb):lsb]; rd = SE32(exword);  RV64: a64 = rs1; lsb = imm5u; exword = a64[(31+lsb):lsb]; rd = SE64(exword);
8	CMIX rd, rs2, rs1, rs3	Conditional Mix	$rd[i] = rs2[i]? rs1[i] : rs3[i];$ (RV32: i=31..0, RV64: i=63..0)
9	INSB rd, rs1, imm2u/imm3u	Insert Byte	RV32: byte_idx = imm2u; rd.B[byte_idx] = rs1.B[0];  RV64: byte_idx = imm3u; rd.B[byte_idx] = rs1.B[0];
10	MADDR32 rd, rs1, tb	Multiply and Add to 32-bit Word	RV32: Mresult = rs1 * rs2; rd = rd + Mresult.W[0];  RV64: Mresult = rs1.W[0] * rs2.W[0]; tresult[31:0] = rd.W[0] + Mresult.W[0]; rd = SE64(tresult[31:0]);
11	MSUBR32 rd, rs1, tb	Multiply and Subtract from 32-bit Word	RV32: Mresult = rs1 * rs2; rd = rd - Mresult.W[0];  RV64: Mresult = rs1.W[0] * rs2.W[0]; tresult[31:0] = rd.W[0] - Mresult.W[0]; rd = SE64(tresult[31:0]);
12	MAX rd, rs1, rs2	Signed Word Maximum	<pre>if (rs1 s&gt;= rs2) {     rd = rs1; } else {     rd = rs2; }</pre>
13	MIN rd, rs1, rs2	Signed Word Minimum	<pre>if (rs1 s&gt;= rs2) {     rd = rs2; } else {     rd = rs1; }</pre>

### 3.5. 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	PADD.W (ADD32) rd, rs1, rs2	SIMD 32-bit Addition	$rd.W[x] = rs1.W[x] + rs2.W[x];$ (RV64: $x=1..0$ )
2	PAADD.W (RADD32) rd, rs1, rs2	SIMD 32-bit Signed Averaging Addition	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[x] = (a33[x] + b33[x]) s\gg 1;$ (RV64: $x=1..0$ )
3	PAADDU.W (URADD32) rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Addition	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[x] = (a33[x] + b33[x]) u\gg 1;$ (RV64: $x=1..0$ )
4	PSADD.W (KADD32) rd, rs1, rs2	SIMD 32-bit Signed Saturating Addition	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[x] = SAT.Q31(a33[x] + b33[x]);$ (RV64: $x=1..0$ )
5	PSADDU.W (UKADD32) rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Addition	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[x] = SAT.U32(a33[x] + b33[x]);$ (RV64: $x=1..0$ )
6	PSUB.W (SUB32) rd, rs1, rs2	SIMD 32-bit Subtraction	$rd.W[x] = rs1.W[x] - rs2.W[x];$ (RV64: $x=1..0$ )
7	PASUB.W (RSUB32) rd, rs1, rs2	SIMD 32-bit Signed Averaging Subtraction	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[x] = (a33[x] - b33[x]) s\gg 1;$ (RV64: $x=1..0$ )
8	PASUBU.W (URSUB32) rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Subtraction	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[x] = (a33[x] - b33[x]) u\gg 1;$ (RV64: $=1..0$ )
9	PSSUB.W (KSUB32) rd, rs1, rs2	SIMD 32-bit Signed Saturating Subtraction	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[x] = SAT.Q31(a33[x] - b33[x]);$ (RV64: $x=1..0$ )

No.	Mnemonic	Instruction	Operation
10	<b>PSSUBU.W</b> (UKSUB32) rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Subtraction	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[x] = SAT.U32(a33[x] - b33[x]);$ (RV64: x=1..0)
11	<b>PAS.WX</b> (CRAS32) rd, rs1, rs2	SIMD 32-bit Cross Add & Sub	$rd.W[1] = rs1.W[1] + rs2.W[0];$ $rd.W[0] = rs1.W[0] - rs2.W[1];$
12	<b>PAAS.WX</b> (RCRAS32) rd, rs1, rs2	SIMD 32-bit Signed Averaging Cross Add & Sub	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = (a33[1] + b33[0]) s\gg 1;$ $rd.W[0] = (a33[0] - b33[1]) s\gg 1;$
13	URCRAS32 rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Cross Add & Sub	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = (a33[1] + b33[0]) u\gg 1;$ $rd.W[0] = (a33[0] - b33[1]) u\gg 1;$
14	<b>PSAS.WX</b> (KCRAS32) rd, rs1, rs2	SIMD 32-bit Signed Saturating Cross Add & Sub	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = SAT.Q31(a33[1] + b33[0]);$ $rd.W[0] = SAT.Q31(a33[0] - b33[1]);$
15	UKCRAS32 rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Cross Add & Sub	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = SAT.U32(a33[1] + b33[0]);$ $rd.W[0] = SAT.U32(a33[0] - b33[1]);$
16	<b>PSA.WX</b> (CRSA32) rd, rs1, rs2	SIMD 32-bit Cross Sub & Add	$rd.W[1] = rs1.W[1] - rs2.W[0];$ $rd.W[0] = rs1.W[0] + rs2.W[1];$
17	<b>PASA.WX</b> (RCRSA32) rd, rs1, rs2	SIMD 32-bit Signed Averaging Cross Sub & Add	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = (a33[1] - b33[0]) s\gg 1;$ $rd.W[0] = (a33[0] + b33[1]) s\gg 1;$
18	URCRSA32 rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Cross Sub & Add	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = (a33[1] - b33[0]) u\gg 1;$ $rd.W[0] = (a33[0] + b33[1]) u\gg 1;$
19	<b>PSSA.WX</b> (KCRSA32) rd, rs1, rs2	SIMD 32-bit Signed Saturating Cross Sub & Add	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = SAT.Q31(a33[1] - b33[0]);$ $rd.W[0] = SAT.Q31(a33[0] + b33[1]);$
20	UKCRSA32 rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Cross Sub & Add	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = SAT.U32(a33[1] - b33[0]);$ $rd.W[0] = SAT.U32(a33[0] + b33[1]);$
21	STAS32 rd, rs1, rs2	SIMD 32-bit Straight Add & Sub	$rd.W[1] = rs1.W[1] + rs2.W[1];$ $rd.W[0] = rs1.W[0] - rs2.W[0];$

No.	Mnemonic	Instruction	Operation
22	RSTAS32 rd, rs1, rs2	SIMD 32-bit Signed Averaging Straight Add & Sub	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = (a33[1] + b33[1]) s\gg 1;$ $rd.W[0] = (a33[0] - b33[0]) s\gg 1;$
23	URSTAS32 rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Straight Add & Sub	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = (a33[1] + b33[1]) u\gg 1;$ $rd.W[0] = (a33[0] - b33[0]) u\gg 1;$
24	KSTAS32 rd, rs1, rs2	SIMD 32-bit Signed Saturating Straight Add & Sub	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = SAT.Q31(a33[1] + b33[1]);$ $rd.W[0] = SAT.Q31(a33[0] - b33[0]);$
25	UKSTAS32 rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Straight Add & Sub	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = SAT.U32(a33[1] + b33[1]);$ $rd.W[0] = SAT.U32(a33[0] - b33[0]);$
26	STSA32 rd, rs1, rs2	SIMD 32-bit Straight Sub & Add	$rd.W[1] = rs1.W[1] - rs2.W[1];$ $rd.W[0] = rs1.W[0] + rs2.W[0];$
27	RSTSA32 rd, rs1, rs2	SIMD 32-bit Signed Averaging Straight Sub & Add	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = (a33[1] - b33[1]) s\gg 1;$ $rd.W[0] = (a33[0] + b33[0]) s\gg 1;$
28	URSTSA32 rd, rs1, rs2	SIMD 32-bit Unsigned Averaging Straight Sub & Add	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = (a33[1] - b33[1]) u\gg 1;$ $rd.W[0] = (a33[0] + b33[0]) u\gg 1;$
29	KSTSA32 rd, rs1, rs2	SIMD 32-bit Signed Saturating Straight Sub & Add	$a33[x] = SE33(rs1.W[x]);$ $b33[x] = SE33(rs2.W[x]);$ $rd.W[1] = SAT.Q31(a33[1] - b33[1]);$ $rd.W[0] = SAT.Q31(a33[0] + b33[0]);$
30	UKSTSA32 rd, rs1, rs2	SIMD 32-bit Unsigned Saturating Straight Sub & Add	$a33[x] = ZE33(rs1.W[x]);$ $b33[x] = ZE33(rs2.W[x]);$ $rd.W[1] = SAT.U32(a33[1] - b33[1]);$ $rd.W[0] = SAT.U32(a33[0] + b33[0]);$

There are 14 SIMD 32-bit shift instructions.

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

No.	Mnemonic	Instruction	Operation
1	SRA32 rd, rs1, rs2	SIMD 32-bit Shift Right Arithmetic	$rd.W[x] = rs1.W[x] s\gg rs2[4:0];$ (RV64: x=1..0)
2	SRAI32 rd, rs1, im5u	SIMD 32-bit Shift Right Arithmetic Immediate	$rd.W[x] = rs1.W[x] s\gg im5u;$ (RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
3	SRA32.u rd, rs1, rs2	SIMD 32-bit Rounding Shift Right Arithmetic	$rd.W[x] = \text{ROUND}(rs1.W[x] \text{ s}>> rs2[4:0]);$ (RV64: $x=1..0$ )
4	SRAI32.u rd, rs1, im5u	SIMD 32-bit Rounding Shift Right Arithmetic Immediate	$rd.W[x] = \text{ROUND}(rs1.W[x] \text{ s}>> im5u);$ (RV64: $x=1..0$ )
5	SRL32 rd, rs1, rs2	SIMD 32-bit Shift Right Logical	$rd.W[x] = rs1.W[x] \text{ u}>> rs2[4:0];$ (RV64: $x=1..0$ )
6	SRLI32 rd, rs1, im5u	SIMD 32-bit Shift Right Logical Immediate	$rd.W[x] = rs1.W[x] \text{ u}>> im5u;$ (RV64: $x=1..0$ )
7	SRL32.u rd, rs1, rs2	SIMD 32-bit Rounding Shift Right Logical	$rd.W[x] = \text{ROUND}(rs1.W[x] \text{ u}>> rs2[4:0]);$ (RV64: $x=1..0$ )
8	SRLI32.u rd, rs1, im5u	SIMD 32-bit Rounding Shift Right Logical Immediate	$rd.W[x] = \text{ROUND}(rs1.W[x] \text{ u}>> im5u);$ (RV64: $x=1..0$ )
9	SLL32 rd, rs1, rs2	SIMD 32-bit Shift Left Logical	$rd.W[x] = rs1.W[x] \text{ << rs2[4:0];}$ (RV64: $x=1..0$ )
10	SLLI32 rd, rs1, im5u	SIMD 32-bit Shift Left Logical Immediate	$rd.W[x] = rs1.W[x] \text{ << im5u};$ (RV64: $x=1..0$ )
11	KSLL32 rd, rs1, rs2	SIMD 32-bit Saturating Shift Left Logical	$rd.W[x] = \text{SAT.Q31}(rs1.W[x] \text{ << rs2[4:0]});$ (RV64: $x=1..0$ )
12	KSLLI32 rd, rs1, im5u	SIMD 32-bit Saturating Shift Left Logical Immediate	$rd.W[x] = \text{SAT.Q31}(rs1.W[x] \text{ << im5u});$ (RV64: $x=1..0$ )
13	KSLRA32 rd, rs1, rs2	SIMD 32-bit Shift Left Logical with Saturation & Shift Right Arithmetic	$a[x] = rs1.W[x];$ $\text{if } (rs2[5:0] \text{ s}< 0)$ $\quad rd.W[x] = a[x] \text{ s}>> -rs2[5:0];$  $\text{if } (rs2[5:0] \text{ s}> 0)$ $\quad rd.W[x] = \text{SAT.Q31}(a[x] \text{ << rs2[5:0]});$  (RV64: $x=1..0$ )

No.	Mnemonic	Instruction	Operation
14	KSLRA32.u rd, rs1, rs2	SIMD 32-bit Shift Left Logical with Saturation & Rounding Shift Right Arithmetic	$a[x] = rs1.W[x];$ if ( $rs2[5:0] \leq 0$ ) $rd.W[x] = \text{ROUND}(a[x] \ll -rs2[5:0]);$  if ( $rs2[5:0] > 0$ ) $rd.W[x] = \text{SAT.Q31}(a[x] \ll rs2[5:0]);$  (RV64: $x=1..0$ )

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

No.	Mnemonic	Instruction	Operation
1	<b>PMIN.W (SMIN32)</b> rd, rs1, rs2	SIMD 32-bit Signed Minimum	$lt[x] = rs1.W[x] \leq rs2.W[x];$ $rd.W[x] = (lt[x])? rs1.W[x] : rs2.W[x];$  (RV64: $x=1..0$ )
2	<b>PMINU.W (UMIN32)</b> rd, rs1, rs2	SIMD 32-bit Unsigned Minimum	$lt[x] = rs1.W[x] \leq rs2.W[x];$ $rd.W[x] = (lt[x])? rs1.W[x] : rs2.W[x];$  (RV64: $x=1..0$ )
3	<b>PMAX.W (SMAX32)</b> rd, rs1, rs2	SIMD 32-bit Signed Maximum	$gt[x] = rs1.W[x] \geq rs2.W[x];$ $rd.W[x] = (gt[x])? rs1.W[x] : rs2.W[x];$  (RV64: $x=1..0$ )
4	<b>PMAXU.W (UMAX32)</b> rd, rs1, rs2	SIMD 32-bit Unsigned Maximum	$gt[x] = rs1.W[x] \geq rs2.W[x];$ $rd.W[x] = (gt[x])? rs1.W[x] : rs2.W[x];$  (RV64: $x=1..0$ )
5	<b>PABS.W (KABS32)</b> rd, rs1	SIMD 32-bit Absolute Value	$rd.W[x] = \text{ABS}(rs1.W[x]);$  (RV64: $x=1..0$ )

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

No.	Mnemonic	Instruction	Operation
1	KHMBB16 rd, rs1, rs2	Multiply the first 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q15 numbers.	$t[x] = rs1.W[x].H[0] \times rs2.W[x].H[0];$ $rd.W[x] = \text{SAT.Q15}(t[x] \ll 15);$  (RV64: $x=1..0$ )

No.	Mnemonic	Instruction	Operation
2	KHMBT16 rd, rs1, rs2	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.	$t[x] = rs1.W[x].H[0] s* rs2.W[x].H[1];$ $rd.W[x] = SAT.Q15(t[x] s>> 15);$ (RV64: x=1..0)
3	KHMTT16 rd, rs1, rs2	Multiply the second 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q15 numbers.	$t[x] = rs1.W[x].H[1] s* rs2.W[x].H[1];$ $rd.W[x] = SAT.Q15(t[x] s>> 15);$ (RV64: x=1..0)
4	KDMBB16 rd, rs1, rs2	Multiply the first 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers.	$t[x] = rs1.W[x].H[0] s* rs2.W[x].H[0];$ $rd.W[x] = SAT.Q31(t[x] << 1);$ (RV64: x=1..0)
5	KDMBT16 rd, rs1, rs2	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.	$t[x] = rs1.W[x].H[0] s* rs2.W[x].H[1];$ $rd.W[x] = SAT.Q31(t[x] << 1);$ (RV64: x=1..0)

No.	Mnemonic	Instruction	Operation
6	KDMTT16 rd, rs1, rs2	Multiply the second 16-bit Q15 elements of 32-bit chunks in two registers and transform the Q30 results into saturated Q31 numbers.	$t[x] = rs1.W[x].H[1] s* rs2.W[x].H[1];$ $rd.W[x] = SAT.Q31(t[x] << 1);$ (RV64: x=1..0)
7	KDMABB16 rd, rs1, rs2	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.	$t[x] = rs1.W[x].H[0] s* rs2.W[x].H[0];$ $res[x] = SAT.Q31(t[x] << 1);$ $rd.W[x] = SAT.Q31(rd.W[x] + res[x]);$ (RV64: x=1..0)
8	KDMABT16 rd, rs1, rs2	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.	$t[x] = rs1.W[x].H[0] s* rs2.W[x].H[1];$ $res[x] = SAT.Q31(t[x] << 1);$ $rd.W[x] = SAT.Q31(rd.W[x] + res[x]);$ (RV64: x=1..0)
9	KDMATT16 rd, rs1, rs2	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.	$t[x] = rs1.W[x].H[1] s* rs2.W[x].H[1];$ $res[x] = SAT.Q31(t[x] << 1);$ $rd.W[x] = SAT.Q31(rd.W[x] + res[x]);$ (RV64: x=1..0)

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

No.	Mnemonic	Instruction	Operation
1	<b>MUL.WEE rd, rs1, rs2</b>	Multiply the first 32-bit elements of two registers.	$rd = rs1.W[0] \text{ s}^* rs2.W[0];$
2	<b>MUL.WEO rd, rs1, rs2</b>	Multiply the first 32-bit element of one register with the second 32-bit element of another register.	$rd = rs1.W[0] \text{ s}^* rs2.W[1];$
3	<b>MUL.WOO rd, rs1, rs2</b>	Multiply the second 32-bit elements of two registers.	$rd = rs1.W[1] \text{ s}^* rs2.W[1];$

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

No.	Mnemonic	Instruction	Operation
1	<b>MACC.WOO</b> (KMABB32) rd, rs1, rs2	Multiply the first 32-bit elements of two registers and the signed multiplication result is added to a third register. ( $64 = 64 + 32 \times 32$ )	$rd = rd + rs1.W[0] * rs2.W[0];$
2	<b>MACC.WEO</b> (KMABT32) rd, rs1, rs2	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. ( $64 = 64 + 32 \times 32$ )	$rd = rd + rs1.W[0] s* rs2.W[1];$
3	<b>MACC.WEE</b> (KMATT32) rd, rs1, rs2	Multiply the second 32-bit elements of two registers and the signed multiplication result is added to a third register. ( $64 = 64 + 32 \times 32$ )	$rd = rd + rs1.W[1] s* rs2.W[1];$

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

No.	Mnemonic	Instruction	Operation
1	PM2ADD.W (KMDA32) rd, rs1, rs2	Multiply the corresponding 32-bit Elements of two registers and add the results. ( $64 = 32 \times 32 + 32 \times 32$ )	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = t1 + t0;$
2	PM2ADD.WX (KMXDA32) rd, rs1, rs2	Multiply the cross-positioned 32-bit elements of two registers and add the signed multiplication results. ( $64 = 32 \times 32 + 32 \times 32$ ) Elements of two	$t01 = rs1.W[0] s^* rs2.W[1];$ $t10 = rs1.W[1] s^* rs2.W[0];$ $rd = t10 + t01;$

No.	Mnemonic	Instruction	Operation
3	KMADA32 rd, rs1, rs2	<p>Multiply the corresponding 32-bit elements of two registers and add the signed multiplication results and a third register with Q63 saturation.</p> $(64 = 64 + 32 \times 32 + 32 \times 32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = SAT.Q63(rd + t1 + t0);$
4	PM2ADDA.WX (KMAXDA32) rd, rs1, rs2	<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> $(64 = 64 + 32 \times 32 + 32 \times 32)$	$t01 = rs1.W[0] s^* rs2.W[1];$ $t10 = rs1.W[1] s^* rs2.W[0];$ $rd = rd + t10 + t01;$
5	KMADS32 rd, rs1, rs2	<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> $(64 = 64 + 32 \times 32 - 32 \times 32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = SAT.Q63(rd + t1 - t0);$

No.	Mnemonic	Instruction	Operation
6	PM2SUBA.W (KMADRS32) rd, rs1, rs2	<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.</p> $(64 = 64 + 32x32 - 32x32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = rd + t0 - t1;$
7	PM2SUBA.WX (KMAXDS32) rd, rs1, rs2	<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> $(64 = 64 + 32x32 - 32x32)$	$t01 = rs1.W[0] s^* rs2.W[1];$ $t10 = rs1.W[1] s^* rs2.W[0];$ $rd = SAT.Q63(rd + t10 - t01);$
8	KMSDA32 rd, rs1, rs2	<p>Multiply the corresponding 32-bit elements of two registers and subtract the signed multiplication results from a third register with Q63 saturation.</p> $(64 = 64 - 32x32 - 32x32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = SAT.Q63(rd - t1 - t0);$

No.	Mnemonic	Instruction	Operation
9	KMSXDA32 rd, rs1, rs2	Multiply the cross-positioned 32-bit elements of two registers and subtract the signed multiplication results from a third register with Q63 saturation.  $(64 = 64 - 32 \times 32 - 32 \times 32)$	$t01 = rs1.W[0] s^* rs2.W[1];$ $t10 = rs1.W[1] s^* rs2.W[0];$ $rd = SAT.Q63(rd - t10 - t01);$
10	SMDS32 rd, rs1, rs2	Multiply the corresponding 32-bit elements of two registers and subtract the bottom signed multiplication result from the top result.  $(64 = 32 \times 32 - 32 \times 32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = t1 - t0;$
11	PM2SUB.W (SMDRS32) rd, rs1, rs2	Multiply the corresponding 32-bit elements of two registers and subtract the odd signed multiplication result from the even result.  $(64 = 32 \times 32 - 32 \times 32)$	$t0 = rs1.W[0] s^* rs2.W[0];$ $t1 = rs1.W[1] s^* rs2.W[1];$ $rd = t0 - t1;$
12	PM2SUB.WX (SMXDS32) rd, rs1, rs2	Multiply the cross-positioned 32-bit elements of two registers and subtract the bottom signed multiplication result from the top result.  $(64 = 32 \times 32 - 32 \times 32)$	$t01 = rs1.W[0] s^* rs2.W[1];$ $t10 = rs1.W[1] s^* rs2.W[0];$ $rd = t10 - t01;$

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

No.	Mnemonic	Instruction	Operation
1	SRAIW.u rd, rs1, imm5u	32-bit Rounding arithmetic shift right immediate	$rd = SE64(ROUND(rs1.W[0] \ s>> imm5u));$

There are four 32-bit packing instructions here.

Table 35. (RV64 Only) 32-bit Packing Instructions

No.	Mnemonic	Instruction	Operation
1	PKBB32 rd, rs1, rs2	Pack two 32-bit data from Bottoms	$rd = CONCAT(rs1.W[0], rs2.W[0]);$
2	PKBT32 rd, rs1, rs2	Pack two 32-bit data Bottom & Top	$rd = CONCAT(rs1.W[0], rs2.W[1]);$
3	PKTB32 rd, rs1, rs2	Pack two 32-bit data Top & Bottom	$rd = CONCAT(rs1.W[1], rs2.W[0]);$
4	PKTT32 rd, rs1, rs2	Pack two 32-bit data from Tops	$rd = CONCAT(rs1.W[1], rs2.W[1]);$

## **Chapter 4. Instructions Duplicated with Other Extensions (hidden for now)**

## **Chapter 5. P Extension Subsets (hidden for now)**

# **Chapter 6. Detailed Instruction Descriptions for Zbpbo Extension (hidden for now)**

# Chapter 7. Detailed Instruction Descriptions for Zpn Extension (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.

## Processing of 64-bit Values in RV32

Some RV32 instructions read or write 64-bit operands with paired 32-bit registers. These paired registers are constrained to a pair of aligned { R<sub>n</sub>, R<sub>n+1</sub> } registers with "n" being an even number. Use of misaligned (odd-numbered) registers for 64-bit operands is *reserved*.

Regardless of endianness, the lower-numbered register holds the low-order bits, and the higher-numbered register holds the high-order bits: e.g., bits 31:0 of a 64-bit operand might be held in register x14, with bits 63:32 of that operand held in x15.

When a 64-bit result is written to x0, the entire write takes no effect: i.e., writing a 64-bit result to x0 does not cause x1 to be written.

When x0 is used as a 64-bit operand, the entire operand is zero—i.e., x1 is not accessed.

## Intrinsic Function Data Type Definition:

Having a fixed-size data type increases code readability and avoids confusion on the value range of an operand. On the other hand, the instructions defined here operate on registers of different sizes. Some operands have a size depending on the XLEN value. Some operands have fixed sizes independent of the XLEN value. To help distinguish them and still conform to the fixed-size principle, the following symbols are defined and used in the intrinsic function prototype descriptions.

- RV32

```
typedef int32_t  intXLEN_t  
typedef uint32_t uintXLEN_t
```

- RV64

```
typedef int64_t  intXLEN_t  
typedef uint64_t uintXLEN_t
```

-

## 7.1. PADD.B (ADD8) (SIMD 8-bit Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PADD.B 0100100	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PADD.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit integer element additions in parallel.

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.

.

## 7.2. PADD.H (ADD16) (SIMD 16-bit Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PADD.H (ADD16) 0100000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PADD.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit integer element additions in parallel.

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.

## 7.3. PMSEQ.B (CMPEQ8) (SIMD 8-bit Integer Compare Equal)

Type: SIMD

Format:

31 25	24 20	19 15	14 12	11 7	6 0
PMSEQ.B 0100111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSEQ.8 Rd, Rs1, Rs2
```

Purpose: Perform 8-bit integer elements equal comparisons in parallel.

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.

.

## 7.4. PMSEQ.H (CMPEQ16) (SIMD 16-bit Integer Compare Equal)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSEQ.H 0100110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSEQ.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit integer elements equal comparisons in parallel.

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

## 7.5. PAS.HX (CRAS16) (SIMD 16-bit Cross Addition & Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PAS.HX 0100010	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PAS.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk in parallel. 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].H[1] = Rs1.W[x].H[1] + Rs2.W[x].H[0];  
Rd.W[x].H[0] = Rs1.W[x].H[0] - Rs2.W[x].H[1];  
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.

## 7.6. PSA.HX (CRSA16) (SIMD 16-bit Cross Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSA.HX 0100011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSA.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk in parallel. 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].H[1] = Rs1.W[x].H[1] - Rs2.W[x].H[0];
Rd.W[x].H[0] = Rs1.W[x].H[0] + Rs2.W[x].H[1];
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.

## 7.7. PABS.H (KABS16) (SIMD 16-bit Unsigned Absolute)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ONEOP 1010110	PABS.H (KABS16) 10001	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PABS.H Rd, Rs1
```

Purpose: Compute the absolute value of 16-bit signed integer elements in parallel.

Description: This instruction calculates the absolute value of 16-bit unsigned integer elements stored in Rs1 and writes the element results to Rd.

Operations:

```
src = Rs1.H[x];
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:

## 7.8. PSADD.B (KADD8) (SIMD 8-bit Signed Saturating Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSADD.B 0001100	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSADD.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit signed integer element saturating additions in parallel.

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 exceed 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:

```
a9[x] = SE9(Rs1.B[x]);  
b9[x] = SE9(Rs2.B[x]);  
res9[x] = a9[x] + b9[x];  
if (res9[x] s> (2^7)-1) {  
    res9[x] = 127;  
    OV = 1;  
} else if (res9[x] s< -2^7) {  
    res9[x] = -128;  
    OV = 1;  
}  
Rd.B[x] = res9[x].B[0];  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.9. PSADD.H (KADD16) (SIMD 16-bit Signed Saturating Addition)

Type: SIMD

**Format:**

31    25	24    20	19    15	14    12	11    7	6    0
PSADD.H 0001000	Rs2	Rs1	000	Rd	OP-P 1110111

**Syntax:**

```
PSADD.H Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element saturating additions in parallel.

**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 exceed 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:**

```
a17[x] = SE17(Rs1.H[x]);
b17[x] = SE17(Rs2.H[x]);
res17[x] = a17[x] + b17[x];
if (res17[x] > (2^15)-1) {
    res17[x] = 32767;
    OV = 1;
} else if (res17[x] < -(2^15)) {
    res17[x] = -32768;
    OV = 1;
}
Rd.H[x] = res17[x].H[0];
for RV32: x=1..0,
for RV64: x=3..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.10. SADD (KADDW) (Signed Addition with Q31 Saturation)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
SADD 0000000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SADD 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:

```
a33 = SE33(Rs1.W[0]);  
b33 = SE33(Rs2.W[0]);  
tmp33 = a33 + b33;  
if (tmp33 > (2^31)-1) {  
    res32 = (2^31)-1;  
    OV = 1;  
} else if (tmp33 < -2^31) {  
    res32 = -2^31;  
    OV = 1;  
} else {  
    res32 = tmp33.W[0];  
}  
Rd = res32; // RV32  
Rd = SE64(res32); // RV64
```

Exceptions: None

Privilege level: All

Note:

## 7.11. PSAS.HX (KCRAS16) (SIMD 16-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSAS.HX 0001010	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSAS.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk in parallel. 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 exceed 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:

```
a17[x] = SE17(Rs1.W[x].H[1]);  
b17[x] = SE17(Rs2.W[x].H[0]);  
c17[x] = SE17(Rs1.W[x].H[0]);  
d17[x] = SE17(Rs2.W[x].H[1]);  
res1 = a17[x] + b17[x];  
res2 = c17[x] - d17[x];  
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].H[1] = res1.H[0];  
Rd.W[x].H[0] = res2.H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

**Privilege level:** All

**Note:**

-

## 7.12. PSSA.HX (KCRSA16) (SIMD 16-bit Signed Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSSA.HX 0001011		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PSSA.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk in parallel. 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 exceed 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:

```
a17[x] = SE17(Rs1.W[x].H[1]);  
b17[x] = SE17(Rs2.W[x].H[0]);  
c17[x] = SE17(Rs1.W[x].H[0]);  
d17[x] = SE17(Rs2.W[x].H[1]);  
res1 = a17[x] - b17[x];  
res2 = c17[x] + d17[x];  
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].H[1] = res1.H[0];  
Rd.W[x].H[0] = res2.H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

**Privilege level:** All

**Note:**

## 7.13. PMULQ.H (KHM16), KHMX16

### 7.13.1. PMULQ.H (KHM16) (SIMD Signed Saturating Q15 Multiply)

### 7.13.2. KHMX16 (SIMD Signed Saturating Crossed Q15 Multiply)

Type: SIMD

Format:

#### PMULQ.H

31 25	24 20	19 15	14 12	11 7	6 0
PMULQ.H 1000011	Rs2	Rs1	000	Rd	OP-P 1110111

#### KHMX16

31 25	24 20	19 15	14 12	11 7	6 0
KHMX16 1001011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMULQ.H (KHM16)
Rd, Rs1, Rs2 KHMX16
Rd, Rs1, Rs2
```

Purpose: Perform Q15xQ15 element multiplications in parallel. The Q30 results are then reduced to Q15 numbers again.

Description: For the "PMULQ.H" instruction, multiply the even 16-bit Q15 content of 32-bit chunks in Rs1 with the even 16-bit Q15 content of 32-bit chunks in Rs2. At the same time, multiply the odd 16-bit Q15 content of 32-bit chunks in Rs1 with the odd 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 "PMULQ.H") {
    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,res32) in [(op1t,op2t,rest), (op1b,op2b,rest)]) {
    if ((0x8000 != aop) || (0x8000 != bop)) {
        mres[31:0] = aop s* bop;
        rshifted[31:0] = mres[31:0] s>> 15;
        res32 = rshifted[31:0];
    } else {
        res32= 0x00007FFF;
        OV = 1;
    }
}
Rd.W[x/2] = concat(rest.H[0], resb.H[0]);
```

```
for RV32: x=0
for RV64: x=0,2
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.14. PMACC.W.HEE (KMABB), PMAC.W.HEO (KMABT), PMAC.W.HOO (KMATT)

### 7.14.1. PMACC.W.HEE (KMABB) (SIMD Signed Multiply Even Halfs & Add)

### 7.14.2. PMAC.W.HEO (KMABT) (SIMD Signed Multiply Even & Odd Halfs & Add)

### 7.14.3. PMACC.W.HOO (KMATT) (SIMD Signed Multiply Odd Halfs & Add)

Type: SIMD

Format:

**PMACC.W.HEE**

31    25	24    20	19    15	14    12	11    7	6    0
PMACC.W.HEE 0101101	Rs2	Rs1	001	Rd	OP-P 1110111

**PMACC.W.HE  
O**

31    25	24    20	19    15	14    12	11    7	6    0
PMACC.W.HEO 0110101	Rs2	Rs1	001	Rd	OP-P 1110111

**PMACC.W.HO  
O**

31    25	24    20	19    15	14    12	11    7	6    0
PMACC.W.HOO 0111101	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PMACC.W.HEE Rd, Rs1, Rs2  
PMAC.W.HEO Rd, Rs1, Rs2  
PMAC.W.HOO 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 is written to the third register.

- PMACC.W.HEE: rd.W[x] + even\*even (per 32-bit element)
- PMAC.W.HEO: rd.W[x] + even\*odd (per 32-bit element)
- PMACC.W.HOO: rd.W[x] + odd\*odd (per 32-bit element)

For the "PMAC.W.HEE" instruction, it multiplies the *even* 16-bit content of 32-bit elements in Rs1 with the *even* 16-bit content of 32-bit elements in Rs2.

For the "PMAC.W.HEO" instruction, it multiplies the *even* 16-bit content of 32-bit elements in Rs1 with the *odd* 16-bit content of 32-bit elements in Rs2.

For the "PMACC.W.HOO" instruction, it multiplies the *odd* 16-bit content of 32-bit elements in Rs1 with the *odd* 16-bit content of 32-bit elements in Rs2.

The multiplication result is added to the content of 32-bit elements in Rd. The result is written to Rd. The 16-bit contents of Rs1 and Rs2 are treated as signed integers.

**Operations:**

```
mul32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[0]; // PMACC.W.HEE  
mul32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[1]; // PMACC.W.HEO  
mul32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[1]; // PMACC.W.HOO
```

```
res33[x] = SE33(Rd.W[x]) + SE33(mul32[x]);  
Rd.W[x] = res33[x].W[0];  
for RV32: x=0  
for RV64: x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.15. MHACC/PMHACC.W (KMMAC), MHRACC/PMHRACC.W (KMMAC.u)

### 7.15.1. MHACC/PMHACC.W (KMMAC) (SIMD MSW Signed Multiply Word and Add)

### 7.15.2. MHRACC/PMHRACC.W (KMMAC.u) (SIMD MSW Signed Multiply Word and Add with Rounding)

Type: SIMD

Format:

MHACC/PMH  
ACC.W

31 25	24 20	19 15	14 12	11 7	6 0
MHACC PMHACC.W 0110000	Rs2	Rs1	001	Rd	OP-P 1110111

MHRACC/PMHRACC.W

31 25	24 20	19 15	14 12	11 7	6 0
MHRACC PMHRACC.W 0111000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
MHACC Rd, Rs1, Rs2
PMHACC.W Rd, Rs1, Rs2
MHRACC Rd, Rs1, Rs2
PMHRACC.W 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 "MHRACC/PMHRACC.W" 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. The results after saturation are written to Rd. The "MHRACC/PMHRACC.W" 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 (“MHRACC” form || “PMHRACC.W” form) {
    Round[x][32:0] = Mres[x][63:31] + 1;
    res33[x] = SE33(Rd.W[x]) + SE33(Round[x][32:1]);
} else {
    res33[x] = SE33(Rd.W[x]) + SE33(Mres[x][63:32]);
}

Rd.W[x] = res33[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.16. PMHACC.W.HE (KMMAWB), KMMAWB.u

### 7.16.1. PMHACC.W.HE (KMMAWB) (SIMD Saturating MSW Signed Multiply Word and Bottom Half and Add)

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

Type: SIMD

Format:

PMHACC.W.H  
E

31 25	24 20	19 15	14 12	11 7	6 0
PMHACC.W.HE 0100011	Rs2	Rs1	001	Rd	OP-P 1110111

KMMAWB.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWB.u 0101011	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PMHACC.W.HE Rd, Rs1, Rs2  
KMMAWB.u Rd, Rs1, Rs2

**Purpose:** Multiply the signed 32-bit integer elements of one register and the *even* 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 *even* 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. The results 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] s* Rs2.W[x].H[0];
if ('.u' form) {
    Round[x][32:0] = Mres[x][47:15] + 1;
    res33[x] = SE33(Rd.W[x]) + SE33(Round[x][32:1]);
} else {
    res33[x] = SE33(Rd.W[x]) + SE33(Mres[x][47:16]);
}
Rd.W[x] = res33[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.17. PMHACC.W.HO (KMMAWT), KMMAWT.u

### 7.17.1. PMHACC.W.HO (KMMAWT) (SIMD MSW Signed Multiply Word and Odd Half and Add)

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

Type: SIMD

Format:

PMHACC.W.H  
O

31 25	24 20	19 15	14 12	11 7	6 0
PMHACC.W.HO 0110011	Rs2	Rs1	001	Rd	OP-P 1110111

KMMAWT.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT.u 0111011	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PMHACC.W.HO Rd, Rs1, Rs2

KMMAWT.u Rd Rs1, Rs2

**Purpose:** Multiply the signed 32-bit integer elements of one register and the signed *odd* 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 *odd* 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. The results 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;
    res33[x] = SE33(Rd.W[x]) + SE33(Round[x][32:1]);
} else {
    res33[x] = SE33(Rd.W[x]) + SE33(Mres[x][47:16]);
}
Rd.W[x] = res33[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.18. PSSUB.B (KSUB8) (SIMD 8-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSSUB.B 0001101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSSUB.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit signed elements saturating subtractions in parallel.

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 exceed 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] = SE9(Rs1.B[x]) - SE9(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].B[0];
for RV32: x=3..0,
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.19. SSUB (KSUBW) (Signed Subtraction with Q31 Saturation)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
SSUB 0000001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SSUB 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:

```
res33 = SE33(Rs1.W[0]) - SE33(Rs2.W[0]);
if (res33 > (2^31)-1) {
    res33 = (2^31)-1;
    OV = 1;
} else if (res33 < -2^31) {
    res33 = -2^31;
    OV = 1
}
Rd = res33.W[0];           // RV32
Rd = SE64(res33.W[0]);   // RV64
```

Exceptions: None

Privilege level: All

Note:

## 7.20. PSSUB.H (KSUB16) (SIMD 16-bit Signed Saturating Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSSUB.H 0001001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSSUB.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer elements saturating subtractions in parallel.

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 exceed 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] = SE17(Rs1.H[x]) - SE17(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].H[0];
for RV32: x=1..0,
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

## 7.21. MULQ/PMULQ.W (KWMMUL), MULQR/PMULQR.W (KWMMUL.u)

### 7.21.1. MULQ/PMULQ.W (KWMMUL) (SIMD Saturating MSW Signed Multiply Word & Double)

### 7.21.2. MULQR/PMULQR.W (KWMMUL.u) (SIMD Saturating MSW Signed Multiply Word & Double with Rounding)

Type: SIMD

Format:

MULQ/PMULQ  
.W

31 25	24 20	19 15	14 12	11 7	6 0
MULQ/PMULQ.W 0110001	Rs2	Rs1	001	Rd	OP-P 1110111

MULQR/PMUL  
QR.W

31 25	24 20	19 15	14 12	11 7	6 0
MULQR PMULQR.W 0111001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
MULQ Rd, Rs1, Rs2
PMULQ.W Rd, Rs1, Rs2
MULQR Rd, Rs1, Rs2
PMULQR.W 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 "MULQR/PMULQR.W" 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 "MULQR/PMULQR.W" 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] s* Rs2.W[x];  
    if ("MULQR" form || "PMULQR.W" 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:**

## 7.22. Need new opcodes (MULR64) (Multiply Word Unsigned to 64-bit Data)

Type: DSP

Sub-extension: Zpsfoperand

Format:

31      25	24      20	19      15	14      12	11      7	6      0
MULR64 1111000	Rs2	Rs1	001	Rd	OP-P 1110111

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 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 = ZE33(Rs1) u* ZE33(Rs2);  
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];  
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
```

RV64:

```
Mresult = ZE33(Rs1.W[0]) u* ZE33(Rs2.W[0]);  
Rd = Mresult[63:0];
```

Exceptions: None

**Privilege level:** All

**Note:** This instruction can be easily generated by a compiler without using the intrinsic function.

## 7.23. Alias for MUL.WEE - MULSR64 (Multiply Word Signed to 64-bit Data)

Type: DSP

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
MULSR64 1110000	Rs2	Rs1	001	Rd	OP-P 1110111

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 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 = Rs1 s* Rs2;  
R[Rd(4,1).1(0)][31:0] = Mresult[63:32];  
R[Rd(4,1).0(0)][31:0] = Mresult[31:0];
```

RV64:

```
Mresult = Rs1.W[0] s* Rs2.W[0];  
Rd = Mresult[63:0];
```

**Exceptions:** None

**Privilege level:** All

**Note:** This instruction can be easily generated by a compiler without using the intrinsic function.

## 7.24. PDIFSUMU.B (PBSAD) (Parallel Byte Sum of Absolute Difference)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PDIFSUMU.B 1111110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PDIFSUMU.B 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(ZE9(Rs1.B[x]) - ZE9(Rs2.B[x]));  
Rd = SUM(ZE_XLEN(absdiff[x])); // overflow ignored  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.25. PDIFSUMAU.B (PBSADA) (Parallel Byte Sum of Absolute Difference Accum)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PDIFSUMAU.B 1111111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PDIFSUMUA.B Rd, Rs1, Rs2
```

Purpose: Calculate the sum of absolute difference of 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(ZE9(Rs1.B[x]) - ZE9(Rs2.B[x]));  
Rd = Rd + SUM(ZE_XLEN(absdiff[x])); // overflow ignored  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note

## 7.26. PAADD.B (RADD8) (SIMD 8-bit Signed Averaging Addition)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAADD.B 0000100		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PAADD.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit signed integer element additions in parallel. The element results are averaged 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:

```
res9[x] = (SE9(Rs1.B[x]) + SE9(Rs2.B[x])) s>> 1;  
Rd.B[x] = res9[x].B[0];  
for RV32: x=3..0,  
for RV64: x=7..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

## 7.27. PAADD.H (RADD16) (SIMD 16-bit Signed Averaging Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PAADD.H 0000000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PAADD.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer element additions in parallel. The results are averaged 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:

```
res17[x] = (SE17(Rs1.H[x]) + SE17(Rs2.H[x])) s>> 1;  
Rd.H[x] = res17[x].H[0];  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFF, Rs2 = 0x7FFF, Rd = 0x7FFF
- Rs1 = 0x8000, Rs2 = 0x8000, Rd = 0x8000
- Rs1 = 0x4000, Rs2 = 0x8000, Rd = 0xE000

## 7.28. AADD (RADDW) (32-bit Signed Averaging Addition)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
AADD 0010000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
AADD Rd, Rs1, Rs2
```

Purpose: Add 32-bit signed integers and the results are averaged 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:

```
res33 = (SE33(Rs1.W[0]) + SE33(Rs2.W[0])) s>> 1;
```

```
Rd = res33.W[0]; // RV32
Rd = SE64(res33.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0xFFFFFFFF, Rs2 = 0xFFFFFFFF, Rd = 0xFFFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x80000000, Rd = 0x80000000
- Rs1 = 0x40000000, Rs2 = 0x80000000, Rd = 0xE0000000

## 7.29. PAAS.HX (RCRAS16) (SIMD 16-bit Signed Averaging Cross Addition & Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PAAS.HX 0000010	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PAAS.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk in parallel. Operands are from crossed positions in 32-bit chunks. The results are averaged 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:

```
res_add17[x] = (SE17(Rs1.W[x].H[1]) + SE17(Rs2.W[x].H[0])) s>> 1;  
res_sub17[x] = (SE17(Rs1.W[x].H[0]) - SE17(Rs2.W[x].H[1])) s>> 1;  
Rd.W[x].H[1] = res_add17[x].H[0];  
Rd.W[x].H[0] = res_sub17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.H" and "PASUB.H" instructions.

## 7.30. PASA.HX (RCRSA16) (SIMD 16-bit Signed Averaging Cross Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PASA.HX 0000011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PASA.HX Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk in parallel. Operands are from crossed positions in 32-bit chunks. The results are averaged 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:

```
res_sub17[x] = (SE17(Rs1.W[x].H[1]) - SE17(Rs2.W[x].H[0])) s>> 1;  
res_add17[x] = (SE17(Rs1.W[x].H[0]) + SE17(Rs2.W[x].H[1])) s>> 1;  
Rd.W[x].H[1] = res_sub17[x].H[0];  
Rd.W[x].H[0] = res_add17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.H" and "PASUB.H" instructions.Intrinsic functions:

## 7.31. PASUB.B (RSUB8) (SIMD 8-bit Signed Averaging Subtraction)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PASUB.B 0000101		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PASUB.B Rd, Rs1, Rs2
```

**Purpose:** Perform 8-bit signed integer element subtractions in parallel. The results are averaged 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:

```
res9[x] = (SE9(Rs1.B[x]) - SE9(Rs2.B[x])) s>> 1;  
Rd.B[x] = res9[x].B[0];  
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

## 7.32. PASUB.H (RSUB16) (SIMD 16-bit Signed Averaging Subtraction)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PASUB.H 0000001		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PASUB.H Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit signed integer element subtractions in parallel. The results are averaged 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:

```
res17[x] = (SE17(Rs1.H[x]) - SE17(Rs2.H[x])) s>> 1;  
Rd.H[x] = res17[x].H[0];  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFF, Rs2 = 0x8000, Rd = 0x7FFF
- Rs1 = 0x8000, Rs2 = 0x7FFF, Rd = 0x8000
- Rs1 = 0x8000, Rs2 = 0x4000, Rd = 0xA000

## 7.33. ASUB (RSUBW) (32-bit Signed Averaging Subtraction)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ASUB 0010001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
ASUB Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit signed integers and the result is averaged 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:

```
res33 = (SE33(Rs1.W[0]) - SE33(Rs2.W[0])) s>> 1;
```

```
Rd = res33.W[0]; // RV32
Rd = SE64(res33.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x80000000, Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x7FFFFFFF, Rd = 0x80000000
- Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0xA0000000

## 7.34. PMSGE.B (SCMPE8) (SIMD 8-bit Signed Compare Greater Than or Equal)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSGE.B 0001111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSGE.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit signed integer elements greater than or equal comparisons in parallel.

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 greater 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] s>= Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.35. PMSGE.H (SCMPE16) (SIMD 16-bit Signed Compare Greater Than or Equal)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PMSGE.H 0001110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSGE.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer elements greater than or equal comparisons in parallel.

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 greater 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] s>= Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

## 7.36. PMSLT.B (SCMPLT8) (SIMD 8-bit Signed Compare Less Than)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSLT.B 0000111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSLT.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit signed integer elements less than comparisons in parallel.

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:

## 7.37. PMSLT.H (SCMPLT16) (SIMD 16-bit Signed Compare Less Than)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSLT.H 0000110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSLT.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer elements less than comparisons in parallel.

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:

## 7.38. PM4ADDA.H (SMALDA), SMALXDA

### 7.38.1. PM4ADDA.H (SMALDA) (Signed Multiply Two Halves and Two Adds 64-bit)

### 7.38.2. SMALXDA (Signed Crossed Multiply Two Halves and Two Adds 64-bit)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

PM4ADDA.H

31    25	24    20	19    15	14    12	11    7	6    0
PM4ADDA.H 1000110	Rs2	Rs1	001	Rd	OP-P 1110111

SMALXDA

31    25	24    20	19    15	14    12	11    7	6    0
SMALXDA 1001110	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PM4ADDA.H Rd, Rs1, Rs2  
SMALXDA Rd, Rs1, Rs2

Purpose: Perform 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.

- PM4ADDA.H: rd pair+ odd\*odd + even\*even (all 32-bit elements)
- SMALXDA: rd 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.

For this instruction, any potential overflow bits beyond 64-bit are discarded and ignored.

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

#### RV64 Description:

For the "PM4ADDA.H" instruction, it multiplies the *even* 16-bit content of the 32-bit elements of Rs1 with the *even* 16-bit content of the 32-bit elements of Rs2 and then adds the result to the result of multiplying the *odd* 16-bit content of the 32-bit elements of Rs1 with the *odd* 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.

For this instruction, any potential overflow bits beyond 64-bit are discarded and ignored.

#### Operations:

##### RV32:

```
// SMALDA
Mres0[31:0] = (Rs1.H[0] s* Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] s* Rs2.H[1]);
// SMALXDA
Mres0[31:0] = (Rs1.H[0] s* Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] s* Rs2.H[0]);
```

```
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
// overflow ignored
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres0[31:0]) + SE64(Mres1[31:0]);
```

##### RV64:

## 7.39. PM2ADDA.W (SMAR64) (Signed Multiply and Add to 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PM2ADDA.W 1000010	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
PM2ADDA.W 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).

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

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

**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.

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

Operations:

- RV32:

```
d_L = CONCAT(Rd(4,1),1'b0); d_H = CONCAT(Rd(4,1),1'b1);
R[d_H].R[d_L] = R[d_H].R[d_L] + (Rs1 s* Rs2); // overflow discarded
```

- RV64:

```
Rd = Rd + (Rs1.W[0] s* Rs2.W[0]) + (Rs1.W[1] s* Rs2.W[1]); // overflow discarded
```

Exceptions: None

**Privilege level:** All

**Note:**

## 7.40. PMAX.B (SMAX8) (SIMD 8-bit Signed Maximum)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMAX.B 1000101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMAX.B Rd, Rs1, Rs2
```

Purpose: Compute the maximum of each 8-bit signed integer element pair in parallel.

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:

## 7.41. PMAX.H (SMAX16) (SIMD 16-bit Signed Maximum)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMAX.H 1000001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMAX.H Rd, Rs1, Rs2
```

Purpose: Compute the maximum of each 16-bit signed integer element pair in parallel.

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:

## 7.42. PMUL.W.HEE (SMBB16), PMUL.W.HEO (SMBT16), PMUL.W.HOO (SMTT16)

### 7.42.1. PMUL.W.HEE (SMBB16) (SIMD Signed Multiply Even Half & Even Half)

### 7.42.2. PMUL.W.HEO (SMBT16) (SIMD Signed Multiply Even Half & Odd Half)

### 7.42.3. PMUL.W.HOO (SMTT16) (SIMD Signed Multiply Odd Half & Odd Half)

**Type:** SIMD

**Format:**

**PMUL.W.HEE**

31    25	24    20	19    15	14    12	11    7	6    0
PMUL.W.HEE 0000100	Rs2	Rs1	001	Rd	OP-P 1110111

**PMUL.W.HEO**

31    25	24    20	19    15	14    12	11    7	6    0
PMUL.W.HEO 0001100	Rs2	Rs1	001	Rd	OP-P 1110111

**PMUL.W.HOO**

31    25	24    20	19    15	14    12	11    7	6    0
PMUL.W.HOO 0010100	Rs2	Rs1	001	Rd	OP-P 1110111

**Syntax:**

PMUL.W.HEE Rd, Rs1, Rs2  
 PMUL.W.HEO Rd, Rs1, Rs2  
 PMUL.W.HOO 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.

- PMUL.W.HEE: W[x].even \* W[x].even
- PMUL.W.HEO: W[x].even \* W[x].odd
- PMUL.W.HOO: W[x].odd \* W[x].odd

**Description:**

For "PMUL.W.HEE" instruction, it multiplies the even 16-bit content of the 32-bit elements of Rs1

with the *even* 16-bit content of the 32-bit elements of Rs2.

For the “PMUL.W.HEO” instruction, it multiplies the *even* 16-bit content of the 32-bit elements of Rs1 with the *odd* 16-bit content of the 32-bit elements of Rs2.

For the “PMUL.W.HOO” instruction, it multiplies the *odd* 16-bit content of the 32-bit elements of Rs1 with the *odd* 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] s* Rs2.W[x].H[0]; // PMUL.W.HEE  
Rd.W[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[1]; // PMUL.W.HEO  
Rd.W[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[1]; // PMUL.W.HOO  
for RV32: x=0,  
for RV64: x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 7.43. PMIN.B (SMIN8) (SIMD 8-bit Signed Minimum)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMIN.B 1000100	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMIN.B Rd, Rs1, Rs2
```

Purpose: Compute the minimum of each 8-bit signed integer element pair in parallel.

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:

## 7.44. PMIN.H (SMIN16) (SIMD 16-bit Signed Minimum)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMIN.H 1000000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMIN.H Rd, Rs1, Rs2
```

Purpose: Compute the minimum of each 16-bit signed integer element pair in parallel.

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:

## 7.45. PMULH.W (SMMUL), MULHR/PMULHR.W (SMMUL.u)

### 7.45.1. PMULH.W (SMMUL) (SIMD MSW Signed Multiply Word)

### 7.45.2. MULHR/PMULHR.W (SMMUL.u) (SIMD MSW Signed Multiply Word with Rounding)

**Type:** SIMD

- PMULH.W: RV32: Replaced with MULH in RV32M.

**Format:**

SMMUL (RV32)

31 25	24 20	19 15	14 12	11 7	6 0
MULH 0000001	Rs2	Rs1	001	Rd	OP 0110011

PMULH.W  
(RV64)

31 25	24 20	19 15	14 12	11 7	6 0
PMULH.W 0100000	Rs2	Rs1	001	Rd	OP-P 1110111

MULHR/PMUL  
HR.W

31 25	24 20	19 15	14 12	11 7	6 0
MULHR PMULHR.W 0101000	Rs2	Rs1	001	Rd	OP-P 1110111

**Syntax:**

PMULH.W Rd, Rs1, Rs2

MULHR Rd, Rs1, Rs2

PMULHR.W 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 "MULHR/PMUL;HR.W" 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 “MULHR/PMULHR.W” 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 for “mulh/RV32” instruction.

#### Operations:

```
Mres[x][63:0] = Rs1.W[x] s* Rs2.W[x];
if ("MULHR" form || "PMULHR.W" 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:**

## 7.46. PMULH.W.HE (SMMWB), SMMWB.u

### 7.46.1. PMULH.W.HE (SMMWB) (SIMD MSW Signed Multiply Word and Bottom Half)

### 7.46.2. SMMWB.u (SIMD MSW Signed Multiply Word and Bottom Half with Rounding)

Type: SIMD

Format:

PMULH.W.HE

31 25	24 20	19 15	14 12	11 7	6 0
PMULH.W.HE 0100010	Rs2	Rs1	001	Rd	OP-P 1110111

SMMWB.u

31 25	24 20	19 15	14 12	11 7	6 0
SMMWB.u 0101010	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PMULH.W.HE Rd, Rs1, Rs2  
SMMWB.u Rd, Rs1, Rs2

Purpose: Multiply the signed 32-bit integer elements of one register and the *even* 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 *even* 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] s* 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:**

## 7.47. PMULH.W.HO (SMMWT), SMMWT.u

### 7.47.1. PMULH.W.HO (SMMWT) (SIMD MSW Signed Multiply Word and Top Half)

### 7.47.2. SMMWT.u (SIMD MSW Signed Multiply Word and Top Half with Rounding)

Type: SIMD

Format:

#### PMULH.W.HO

31 25	24 20	19 15	14 12	11 7	6 0
PMUH.WO 0110010	Rs2	Rs1	001	Rd	OP-P 1110111

#### SMMWT.u

31 25	24 20	19 15	14 12	11 7	6 0
SMMWT.u 0111010	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

PMULH.W.HO Rd, Rs1, Rs2

SMMWT.u Rd, Rs1, Rs2

Purpose: Multiply the signed 32-bit integer elements of one register and the *odd* 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 *odd* 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] s* 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:**

## 7.48. PSUB.B (SUB8) (SIMD 8-bit Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSUB.B 0100101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSUB.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit integer element subtractions in parallel.

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.

## 7.49. PSUB.H (SUB16) (SIMD 16-bit Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSUB.H 0100001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSUB.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit integer element subtractions in parallel.

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.

## 7.50. SUBD (SUB64) (64-bit Subtraction)

Type: RV32 Only

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
SUBD 1100001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SUBD 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  $2d+1$ .

**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];
```

Exceptions: None

Privilege level: All

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

## 7.51. PMSGEU.B (UCMPE8) (SIMD 8-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PMSGEU.B 0011111		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PMSGEU.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit unsigned integer elements greater than or equal comparisons in parallel.

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 greater 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] u>= Rs2.B[x])? 0xff : 0x0;  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.52. PMSGEU.H (UCMPL16) (SIMD 16-bit Unsigned Compare Less Than & Equal)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSGEU.H 0011110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSGEU.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit unsigned integer elements less than & equal comparisons in parallel.

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] u<= Rs2.H[x])? 0xffff : 0x0;  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

## 7.53. PMSLTU.B (UCMPLT8) (SIMD 8-bit Unsigned Compare Less Than)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PMSLTU.B 0010111		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PMSLTU.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit unsigned integer elements less than comparisons in parallel.

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:

## 7.54. PMSLTU.H (UCMPLT16) (SIMD 16-bit Unsigned Compare Less Than)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMSLTU.H 0010110	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMSLTU.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit unsigned integer elements less than comparisons in parallel.

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:

## 7.55. PSADDU.B (UKADD8) (SIMD 8-bit Unsigned Saturating Addition)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSADDU.B 0011100		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PSADDU.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit unsigned integer element saturating additions in parallel.

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 exceed 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] = ZE9(Rs1.B[x]) + ZE9(Rs2.B[x]);
if (res[x] > (2^8)-1) {
    res[x] = (2^8)-1;
    OV = 1;
}
Rd.B[x] = res[x].B[0];
for RV32: x=3..0,
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.56. PSADDU.H (UKADD16) (SIMD 16-bit Unsigned Saturating Addition)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSADDU.H 0011000		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PSADDU.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit unsigned integer element saturating additions in parallel.

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 exceed 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] = ZE17(Rs1.H[x]) + ZE17(Rs2.H[x]);
if (res[x] > (2^16)-1) {
    res[x] = (2^16)-1;
    OV = 1;
}
Rd.H[x] = res[x].H[0];
for RV32: x=1..0,
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

## 7.57. SADDU (UKADDW) (Unsigned Addition with U32 Saturation)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
SADDU 0001000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SADDU 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:

```
res33 = ZE33(Rs1.W[0]) + ZE33(Rs2.W[0]);
if (res33 > (2^32)-1) {
    res33 = (2^32)-1;
    OV = 1;
}
Rd = res33.W[0];           // RV32
Rd = SE64(res33.W[0]);   // RV64
```

Exceptions: None

Privilege level: All

Note:

## 7.58. PSSUBU.B (UKSUB8) (SIMD 8-bit Unsigned Saturating Subtraction)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSSUBU.B 0011101		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PSSUBU.B Rd, Rs1, Rs2
```

Purpose: Perform 8-bit unsigned integer elements saturating subtractions in parallel.

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 exceed 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] = ZE9(Rs1.B[x]) - ZE9(Rs2.B[x]);
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.B[x] = res[x].B[0];
for RV32: x=3..0,
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

## 7.59. PSSUBU.H (UKSUB16) (SIMD 16-bit Unsigned Saturating Subtraction)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSSUBU.H 0011001		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PSSUBU.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit unsigned integer elements saturating subtractions in parallel.

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 exceed 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] = ZE17(Rs1.H[x]) - ZE17(Rs2.H[x]);
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.H[x] = res[x].H[0];
for RV32: x=1..0,
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

## 7.60. SSUBU (UKSUBW) (Unsigned Subtraction with U32 Saturation)

Type: DSP

Format:

31      25	24      20	19      15	14      12	11      7	6      0
SSUBU 0001001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SSUBU 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:

```
aop33 = ZE33(Rs1.W[0]);  
bop33 = ZE33(Rs2.W[0]);  
res33 = aop33 - bop33;  
if (res33 < 0) {  
    res33 = 0;  
    OV = 1;  
}  
Rd = res33.W[0]; // RV32  
Rd = SE64(res33.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

## 7.61. PMAXU.B (UMAX8) (SIMD 8-bit Unsigned Maximum)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PMAXU.B 1001101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMAXU.B Rd, Rs1, Rs2
```

Purpose: Compute the maximum of each 8-bit unsigned integer element pair in parallel.

Description: This instruction compares the 8-bit unsigned integer elements in Rs1 with the corresponding 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:

## 7.62. PMAXU.H (UMAX16) (SIMD 16-bit Unsigned Maximum)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PMAXU.H 1001001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMAXU.H Rd, Rs1, Rs2
```

Purpose: Compute the maximum of each 16-bit unsigned integer element pair in parallel.

Description: This instruction compares the 16-bit unsigned integer elements in Rs1 with the corresponding 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:

```
uint16x4_t __rv_v_umax16(uint16x4_t a, uint16x4_t b);
```

## 7.63. PSMINU.B (UMIN8) (SIMD 8-bit Unsigned Minimum)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PSMINU.B 1001100	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PSMINU.B Rd, Rs1, Rs2
```

Purpose: Compute the minimum of each 8-bit unsigned integer element pair in parallel.

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] <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:

## 7.64. PMINU.H (UMIN16) (SIMD 16-bit Unsigned Minimum)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PMINU.H 1001000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PMINU.H Rd, Rs1, Rs2
```

Purpose: Compute the minimum of each 16-bit unsigned integer element pair in parallel.

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] <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:

## 7.65. PAADDU.B (URADD8) (SIMD 8-bit Unsigned Averaging Addition)

Type: SIMD

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAADDU.B 0010100		Rs2		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
PAADDU.B Rd, Rs1, Rs2
```

**Purpose:** Perform 8-bit unsigned integer element additions in parallel. The results are averaged 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 9-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res9[x] = (ZE9(Rs1.B[x]) + ZE9(Rs2.B[x])) u>> 1;  
Rd.B[x] = res9[x].B[0];  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7F, Rs2 = 0x7F, Rd = 0x7F
- Rs1 = 0x80, Rs2 = 0x80, Rd = 0x80
- Rs1 = 0x40, Rs2 = 0x80, Rd = 0x60

## 7.66. PAADDU.H (URADD16) (SIMD 16-bit Unsigned Averaging Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PAADDU.H 0010000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PAADDU.H Rd, Rs1, Rs2
```

Purpose: Perform 16-bit unsigned integer element additions in parallel. The results are averaged 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 17-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res17[x] = (ZE17(Rs1.H[x]) + ZE17(Rs2.H[x])) u>> 1;  
Rd.H[x] = res17[x].H[0];  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFF, Rs2 = 0x7FFF Rd = 0x7FFF
- Rs1 = 0x8000, Rs2 = 0x8000 Rd = 0x8000
- Rs1 = 0x4000, Rs2 = 0x8000 Rd = 0x6000

## 7.67. AADDU (URADDW) (32-bit Unsigned Averaging Addition)

Type: DSP

Format:

31      25	24      20	19      15	14      12	11      7	6      0
AADDU 0011000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
AADDU Rd, Rs1, Rs2
```

Purpose: Add 32-bit unsigned integers and the results are averaged 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 33-bit result is first right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

```
res33 = (ZE33(Rs1.W[0]) + ZE33(Rs2.W[0])) u>> 1;
```

```
Rd = res33.W[0]; // RV32  
Rd = SE64(res33.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x7FFFFFFF Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x80000000 Rd = 0x80000000
- Rs1 = 0x40000000, Rs2 = 0x80000000 Rd = 0x60000000

## 7.68. PASUBU.B (URSUB8) (SIMD 8-bit Unsigned Averaging Subtraction)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PASUBU.B 0010101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PASUBU.B Rd, Rs1, Rs2
```

**Purpose:** Perform 8-bit unsigned integer element subtractions in parallel. The results are averaged 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 9-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res9[x] = (ZE9(Rs1.B[x]) - ZE9(Rs2.B[x])) u>> 1;  
Rd.B[x] = res9[x].B[0];  
for RV32: x=3..0,  
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7F, Rs2 = 0x80, Rd = 0xFF
- Rs1 = 0x80, Rs2 = 0x7F, Rd = 0x00
- Rs1 = 0x80, Rs2 = 0x40, Rd = 0x20
- Rs1 = 0x81, Rs2 = 0x01, Rd = 0x40

## 7.69. PASUBU.H (URSUB16) (SIMD 16-bit Unsigned Averaging Subtraction)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
PASUBU.H 0010001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PASUBU.H Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit unsigned integer element subtractions in parallel. The results are averaged 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 17-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res17[x] = (ZE17(Rs1.H[x]) - ZE17(Rs2.H[x])) u>> 1;  
Rd.H[x] = res17[x].H[0];  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFF, Rs2 = 0x8000, Rd = 0xFFFF
- Rs1 = 0x8000, Rs2 = 0x7FFF, Rd = 0x0000
- Rs1 = 0x8000, Rs2 = 0x4000, Rd = 0x2000
- Rs1 = 0x8001, Rs2 = 0x0001, Rd = 0x4000

## 7.70. ASUBU (URSUBW) (32-bit Unsigned Averaging Subtraction)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ASUBU 0011001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
ASUBU Rd, Rs1, Rs2
```

Purpose: Subtract 32-bit unsigned integers and the result is averaged to avoid overflow or saturation.

Description: This instruction subtracts the first 32-bit unsigned integer in Rs2 from the first 32-bit unsigned integer in Rs1. The 33-bit result is first right-shifted by 1 bit and then sign-extended and written to Rd.

Operations:

```
res33 = (ZE33(Rs1.W[0]) - ZE33(Rs2.W[0])) u>> 1;
```

```
Rd = res33.W[0]; // RV32
Rd = SE64(res33.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0xFFFFFFFF, Rs2 = 0x80000000, Rd = 0xFFFFFFFF
- Rs1 = 0x80000000, Rs2 = 0xFFFFFFFF, Rd = 0x00000000
- Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0x20000000
- Rs1 = 0x80000001, Rs2 = 0x00000001, Rd = 0x40000000

# **Chapter 8. Detailed Instruction Descriptions (RV64 Only)**

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

## 8.1. PADD.W (ADD32) (SIMD 32-bit Addition)

Type: SIMD (RV64 Only)

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PADD.W 0100000	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
PADD.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit integer element additions in parallel.

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.

## 8.2. PAS.WX (CRAS32) (SIMD 32-bit Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAS.WX 0100010		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PAS.WX Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk in parallel. 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.

## 8.3. PSA.WX (CRSA32) (SIMD 32-bit Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSA.WX 0100011		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PSA.WX Rd, Rs1, Rs2
```

Purpose: Perform 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk in parallel. 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.

## 8.4. PABS.W (KABS32) (Scalar 32-bit Absolute Value)

Type: DSP (RV64 Only)

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ONEOP 1010110	PABS.W (KABS32) 10010	Rs1	000	Rd	OP-P 1110111

Syntax:

```
PABS.W Rd, Rs1
```

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

Description: This instruction calculates the absolute value of signed 32-bit integer elements stored in Rs1. The results are written to Rd.

Operations:

```
if (Rs1.W[x] >= 0) {  
    res[x] = Rs1.W[x];  
} else {  
    res[x] = -Rs1.W[x];  
}  
Rd.W[x] = res[x];  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.5. PSADD.W (KADD32) (SIMD 32-bit Signed Saturating Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSADD.W 0001000		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PSADD.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit signed integer element saturating additions in parallel.

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 exceed 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:

```
res33[x] = SE33(Rs1.W[x]) + SE33(Rs2.W[x]);
if (res33[x] > (2^31)-1) {
    res33[x] = (2^31)-1;
    OV = 1;
} else if (res33[x] < -2^31) {
    res33[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res33[x].W[0];
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.6. PSAS.WX (KCRAS32) (SIMD 32-bit Signed Saturating Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

**Format:**

31      25	24      20	19      15	14      12	11      7	6      0
PSAS.WX 0001010	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

```
PSAS.WX Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk in parallel. 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 exceed 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] = SE33(Rs1.W[1]) + SE33(Rs2.W[0]);
res[0] = SE33(Rs1.W[0]) - SE33(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].W[0];
Rd.W[0] = res[0].W[0];
for RV64, x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.7. PSSA.WX (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
PSSA.WX 0001011	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

PSSA.WX Rd, Rs1, Rs2

**Purpose:** Perform 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk in parallel. 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 exceed 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] = SE33(Rs1.W[1]) - SE33(Rs2.W[0]);
res[0] = SE33(Rs1.W[0]) + SE33(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].W[0];
Rd.W[0] = res[0].W[0];
for RV64, x=1..0
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.8. MACC.WEE (KMABB32), MACC.WEO (KMABT32), MACC.WOO (KMATT32)

### 8.8.1. MACC.WEE (KMABB32) (Signed Multiply Even Words & Add)

### 8.8.2. MACC.WEO (KMABT32) (Signed Multiply Even & Odd Words & Add)

### 8.8.3. MACC.WOO (KMATT32) (Signed Multiply Odd Words & Add)

Type: DSP (RV64 Only)

**Format:**

**MACC.WEE**

31    25	24    20	19    15	14    12	11    7	6    0
MACC.WEE 0101101	Rs2	Rs1	010	Rd	OP-P 1110111

**MAC.WEO**

31    25	24    20	19    15	14    12	11    7	6    0
MAC.WEO 0110101	Rs2	Rs1	010	Rd	OP-P 1110111

**MAC.WOO**

31    25	24    20	19    15	14    12	11    7	6    0
MACC.WOO 0111101	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

MACC.WEE Rd, Rs1, Rs2  
 MACC.WEO Rd, Rs1, Rs2  
 MACC.WOO 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 is written to the third register.

- MACC.WEE: rd + even\*even
- MACC.WEO: rd + even\*odd
- MACC.WOO: rd + odd\*odd

**Description:**

For the "MACC.WEE" instruction, it multiplies the *even* 32-bit element in Rs1 with the *even* 32-bit element in Rs2.

For the "MACC.WEO" instruction, it multiplies the *even* 32-bit element in Rs1 with the *odd* 32-bit element in Rs2.

For the "MACC.WOO" instruction, it multiplies the *odd* 32-bit element in Rs1 with the *odd* 32-bit element in Rs2.

The multiplication result is added to the content of 64-bit data in Rd. The result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

**Operations:**

```
res65 = SE65(Rd) + SE65(Rs1.W[0] s* Rs2.W[0]); // MACC.WEE  
res65 = SE65(Rd) + SE65(Rs1.W[0] s* Rs2.W[1]); // MACC.WEO  
res65 = SE65(Rd) + SE65(Rs1.W[1] s* Rs2.W[1]); // MACC.WOO  
Rd = res65.D[0];
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.9. KMADA32, PM2ADDA.WX (KMAXDA32)

### 8.9.1. KMADA32 (Saturating Signed Multiply Two Words and Two Adds)

### 8.9.2. PM2ADDA.WX (KMAXDA32) (Saturating Signed Crossed Multiply Two Words and Two Adds)

Type: DSP (RV64 Only)

Format:

KMADA32

31    25	24    20	19    15	14    12	11    7	6    0
KMAR64 1001010	Rs2	Rs1	001	Rd	OP-P 1110111

- An alias for "KMAR64" instruction

PM2ADDA.WX

31    25	24    20	19    15	14    12	11    7	6    0
PM2ADDA.WX 0100101	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
KMADA32 Rd, Rs1, Rs2    # pseudo mnemonic
PM2ADDA.WX Rd, Rs1, Rs2
```

Purpose: Perform 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
- PM2ADDA.WX: rd + odd\*even + even\*odd

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 for 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 exceeds 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:**

```
res66 = SE66(Rd) + SE66(Rs1.W[1] * Rs2.w[1]) + SE66(Rs1.W[0] * Rs2.W[0]); // KMADA32  
res66 = SE66(Rd) + SE66(Rs1.W[1] * Rs2.W[0]) + SE66(Rs1.W[0] * Rs2.W[1]); // PM2ADDA.WX  
Rd = res66.D[0];
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.10. PM2ADD.W (KMDA32), PM2ADD.WX (KMXDA32)

### 8.10.1. PM2ADD.W (KMDA32) (Signed Multiply Two Words and Add)

### 8.10.2. PM2ADD.WX (KMXDA32) (Signed Crossed Multiply Two Words and Add)

**Type:** DSP (RV64 Only)

**Format:**

**PM2ADD.W**

31    25	24    20	19    15	14    12	11    7	6    0
PM2ADD.W 0011100	Rs2	Rs1	010	Rd	OP-P 1110111

**PM2ADD.WX**

31    25	24    20	19    15	14    12	11    7	6    0
PM2ADD.WX 0011101	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

PM2ADD.W Rd, Rs1, Rs2  
PM2ADD.WX Rd, Rs1, Rs2

**Purpose:** Perform two signed 32-bit multiplications from the 32-bit element of two registers, and then adds the two 64-bit results together.

- PM2ADD.W: odd\*odd + even\*even
- PM2ADD.WX: odd\*even + even\*odd

**Description:**

For the "PM2ADD.W" instruction, it multiplies the even 32-bit element of Rs1 with the even 32-bit element of Rs2 and then adds the result to the result of multiplying the odd 32-bit element of Rs1 with the odd 32-bit element of Rs2.

For the "PM2ADD.WX" instruction, it multiplies the even 32-bit element of Rs1 with the odd 32-bit element of Rs2 and then adds the result to the result of multiplying the odd 32-bit element of Rs1 with the even 32-bit element of Rs2.

The result is written to Rd. The 32-bit contents are treated as signed integers.

**Operations:**

```
Rd = (Rs1.W[1] s* Rs2.W[1]) + (Rs1.W[0] s* Rs2.W[0]); // PM2ADD.W  
Rd = (Rs1.W[1] s* Rs2.W[0]) + (Rs1.W[0] s* Rs2.W[1]); // PM2ADD.WX
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## **8.11. KMADS32, PM2SUBA.W (KMADRS32), PM2SUBA.WX (KMAXDS32)**

### **8.11.1. KMADS32 (Saturating Signed Multiply Two Words & Subtract & Add)**

### **8.11.2. PM2SUBA.W (KMADRS32) (Signed Multiply Two Words & Reverse Subtract & Add)**

### **8.11.3. PM2SUBA.WX (KMAXDS32) (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	OP-P 1110111

**PM2SUBA.W**

31    25	24    20	19    15	14    12	11    7	6    0
PM2SUBA.W 0110110	Rs2	Rs1	010	Rd	OP-P 1110111

**PM2SUBA.WX**

31    25	24    20	19    15	14    12	11    7	6    0
PM2SUBA.WX 0111110	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

```
KMADS32 Rd, Rs1, Rs2
PM2SUBA.W Rd, Rs1, Rs2
PM2SUBA.WX Rd, Rs1, Rs2
```

**Purpose:** Perform 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)
- PM2SUBA.W: rd + (even\*even - odd\*odd)
- PM2SUBA.WX: rd + (odd\*even - even\*odd)

### 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 "PM2SUBA.W" instruction, it multiplies the odd 32-bit element in Rs1 with the odd 32-bit element in Rs2 and then subtracts the result from the result of multiplying the even 32-bit element in Rs1 with the even 32-bit element in Rs2.

For the "PM2SUBA.WX" instruction, it multiplies the even 32-bit element in Rs1 with the odd 32-bit element in Rs2 and then subtracts the result from the result of multiplying the odd 32-bit element in Rs1 with the even 32-bit element in Rs2.

The subtraction result is then added to the content of 64-bit data in Rd. The 64-bit result is written to Rd. The 32-bit contents of Rs1 and Rs2 are treated as signed integers.

### Operations:

```
res66 = SE66(Rd) + SE66(Rs1.W[1] * Rs2.W[1]) - SE66(Rs1.W[0] * Rs2.W[0]); // KMADS32  
res66 = SE66(Rd) + SE66(Rs1.W[0] * Rs2.W[0]) - SE66(Rs1.W[1] * Rs2.W[1]); // PM2SUBA.W  
res66 = SE66(Rd) + SE66(Rs1.W[1] * Rs2.W[0]) - SE66(Rs1.W[0] * Rs2.W[1]); // PM2SUBA.WX  
Rd = res66.D[0];
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.12. PSSUB.W (KSUB32) (SIMD 32-bit Signed Saturating Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSSUB.W 0001001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PSSUB.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit signed integer elements saturating subtractions in parallel.

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 exceed 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] = SE33(Rs1.W[x]) - SE33(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].W[0];
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.13. PAADD.W (RADD32) (SIMD 32-bit Signed Averaging Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAADD.W 0000000		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PAADD.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit signed integer element additions in parallel. The results are averaged 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:

```
res33[x] = (SE33(Rs1.W[x]) + SE33(Rs2.W[x])) s>> 1;  
Rd.W[x] = res33[x].W[0];  
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

## 8.14. PAAS.WX (RCRAS32) (SIMD 32-bit Signed Averaging Cross Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAAS.WX 0000010		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PAAS.WX Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk in parallel. Operands are from crossed 32-bit elements. The results are averaged 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:

```
res_add33 = (SE33(Rs1.W[1]) + SE33(Rs2.W[0])) s>> 1;  
res_sub33 = (SE33(Rs1.W[0]) - SE33(Rs2.W[1])) s>> 1;  
Rd.W[1] = res_add33.W[0];  
Rd.W[0] = res_sub33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.W" and "PASUB.W" instructions.

## 8.15. PASA.WX (RCRSA32) (SIMD 32-bit Signed Averaging Cross Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PASA.WX 0000011		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PASA.WX Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk in parallel. Operands are from crossed 32-bit elements. The results are averaged 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:

```
res_sub33 = (SE33(Rs1.W[1]) - SE33(Rs2.W[0])) s>> 1;  
res_add33 = (SE33(Rs1.W[0]) + SE33(Rs2.W[1])) s>> 1;  
Rd.W[1] = res_sub33.W[0];  
Rd.W[0] = res_add33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.W" and "PASUB.W" instructions.

## 8.16. PASUB.W (RSUB32) (SIMD 32-bit Signed Averaging Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PASUB.W 0000001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PASUB.W Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element subtractions in parallel. The results are averaged 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:

```
res33[x] = (SE33(Rs1.W[x]) - SE33(Rs2.W[x])) s>> 1;  
Rd.W[x] = res33[x].W[0];  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0xFFFFFFFF, Rs2 = 0x80000000, Rd = 0x7FFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x7FFFFFFF, Rd = 0x80000000
- Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0xA0000000

## 8.17. PMAX.W (SMAX32) (SIMD 32-bit Signed Maximum)

Type: SIMD (RV64 Only)

**Format:**

31    25	24    20	19    15	14    12	11    7	6    0
PMAX.W 1001001	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

```
PMAX.W Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer elements finding maximum operations in parallel.

**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:**

## 8.18. MUL.WEE (SMBB32), MUL.WEO (SMBT32), MUL.WOO (SMTT32)

### 8.18.1. MUL.WEE (SMBB32) (Signed Multiply Even Word & Even Word)

### 8.18.2. MUL.WEO (SMBT32) (Signed Multiply Even Word & Odd Word)

### 8.18.3. MUL.WOO (SMTT32) (Signed Multiply Odd Word & Odd Word)

Type: DSP (RV64 Only)

**Format:**

MUL.WEE

31    25	24    20	19    15	14    12	11    7	6    0
MUL.WEE 1110000	Rs2	Rs1	001	Rd	OP-P 1110111

- An alias for “MULSR64” instruction

MUL.WEO

31    25	24    20	19    15	14    12	11    7	6    0
MUL.WEO 0001100	Rs2	Rs1	010	Rd	OP-P 1110111

MUL.WOO

31    25	24    20	19    15	14    12	11    7	6    0
MUL.HOO 0010100	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

MUL.WEE Rd, Rs1, Rs2      # pseudo mnemonic

MUL.WEO Rd, Rs1, Rs2

MUL.WOO 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.

- MUL.WEE: even\*even
- MUL.WEO: even\*odd
- MUL.WOO: odd\*odd

**Description:**

For the “MUL.WEE” instruction, it multiplies the *even* 32-bit element of Rs1 with the *even* 32-bit element of Rs2. It is actually an alias for “MULSR64” instruction.

For the “MUL.WEO” instruction, it multiplies the *even* 32-bit element of Rs1 with the *odd* 32-bit element of Rs2.

For the “MUL.WOO” instruction, it multiplies the *odd* 32-bit element of Rs1 with the *odd* 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] s* Rs2.W[0]; // MUL.WEE
res = Rs1.W[0] s* Rs2.w[1]; // MUL.WEO
res = Rs1.W[1] s* Rs2.W[1]; // MULWOO
Rd = res;
```

**Exceptions:** None

**Privilege level:** All

**Note:**

## 8.19. SMDS32, PM2SUB.W (SMDRS32), PM2SUB.WX (SMXDS32)

### 8.19.1. SMDS32 (Signed Multiply Two Words and Subtract)

### 8.19.2. PM2SUB.W (SMDRS32) (Signed Multiply Two Words and Reverse Subtract)

### 8.19.3. PM2SUB.WX (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	OP-P 1110111

PM2SUB.W

31    25	24    20	19    15	14    12	11    7	6    0
PM2SUB.W 0110100	Rs2	Rs1	010	Rd	OP-P 1110111

PM2SUB.WX

31    25	24    20	19    15	14    12	11    7	6    0
PM2SUB.WX 0111100	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
SMDS32 Rd, Rs1, Rs2
PM2SUB.W Rd, Rs1, Rs2
PM2SUB.WX Rd, Rs1, Rs2
```

Purpose: Perform 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
- PM2SUB.W (SMDRS32): even\*even - odd\*odd
- PM2SUB.WX: odd\*even - even\*odd

## 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 "PM2SUB.W" instruction, it multiplies the *odd* 32-bit element of Rs1 with the *odd* 32-bit element of Rs2 and then subtracts the result from the result of multiplying the *even* 32-bit element of Rs1 with the *even* 32-bit element of Rs2.

For the "PM2SUB.WX" instruction, it multiplies the *even* 32-bit element of Rs1 with the *odd* 32-bit element of Rs2 and then subtracts the result from the result of multiplying the *odd* 32-bit element of Rs1 with the *even* 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:

```
Rd = (Rs1.W[1] s* Rs2.W[1]) - (Rs1.W[0] s* Rs2.W[0]); // SMDS32  
Rd = (Rs1.W[0] s* Rs2.W[0]) - (Rs1.W[1] s* Rs2.W[1]); // PM2SUB.W  
Rd = (Rs1.W[1] s* Rs2.W[0]) - (Rs1.W[0] s* Rs2.W[1]); // PM2SUB.WX
```

**Exceptions:** None

**Privilege level:** All

## Note:

- Usage domain: Complex, Statistics, Transform

## 8.20. PMIN.W (SMIN32) (SIMD 32-bit Signed Minimum)

Type: SIMD (RV64 Only)

**Format:**

31    25	24    20	19    15	14    12	11    7	6    0
PMIN.W 1001000	Rs2	Rs1	010	Rd	OP-P 1110111

**Syntax:**

```
PMIN.W Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer elements finding minimum operations in parallel.

**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:**

- Required:

```
int64_t __rv_smin32(int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_smin32(int32x2_t a, int32x2_t b);
```

## 8.21. PSUB.W (SUB32) (SIMD 32-bit Subtraction)

Type: DSP (RV64 Only)

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PSUB.W 0100001	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
PSUB.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit integer element subtractions in parallel.

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.

## 8.22. PSADDU.W (UKADD32) (SIMD 32-bit Unsigned Saturating Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSADDU.W 0011000		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PSADDU.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit unsigned integer element saturating additions in parallel.

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 exceed 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:

```
res33[x] = ZE33(Rs1.W[x]) + ZE33(Rs2.W[x]);
if (res33[x] > (2^32)-1) {
    res33[x] = (2^32)-1;
    OV = 1;
}
Rd.W[x] = res33[x].W[0];
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.23. PSSUBU.W (UKSUB32) (SIMD 32-bit Unsigned Saturating Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PSSUBU.W 0011001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PSSUBU.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit unsigned integer elements saturating subtractions in parallel.

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 exceed 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] = ZE33(Rs1.W[x]) - ZE33(Rs2.W[x]);
if (res[x] < 0) {
    res[x] = 0;
    OV = 1;
}
Rd.W[x] = res[x].W[0];
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.24. PMAXU.W (UMAX32) (SIMD 32-bit Unsigned Maximum)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PMAXU.W 1010001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PMAXU.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit unsigned integer elements finding maximum operations in parallel.

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:

## 8.25. PMINU.W (UMIN32) (SIMD 32-bit Unsigned Minimum)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PMINU.W 1010000		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PMINU.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit unsigned integer elements finding minimum operations in parallel.

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] < u Rs2.W[x])? Rs1.W[x] : Rs2.W[x];  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## 8.26. PAADDU.W (URADD32) (SIMD 32-bit Unsigned Averaging Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PAADDU.W 0010000		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PAADDU.W Rd, Rs1, Rs2
```

Purpose: Perform 32-bit unsigned integer element additions in parallel. The results are averaged 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 33-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res33[x] = (ZE33(Rs1.W[x]) + ZE33(Rs2.W[x])) u>> 1;  
Rd.W[x] = res33[x].W[0];  
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 = 0x60000000

## 8.27. PASUBU.W (URSUB32) (SIMD 32-bit Unsigned Averaging Subtraction)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
PASUBU.W 0010001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
PASUBU.W Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit unsigned integer element subtractions in parallel. The results are averaged 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 33-bit results are first right-shifted by 1 bit and then written to Rd.

Operations:

```
res33[x] = (ZE33(Rs1.W[x]) - ZE33(Rs2.W[x])) u>> 1;  
Rd.W[x] = res33[x].W[0];  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Examples:

- Rs1 = 0x7FFFFFFF, Rs2 = 0x80000000, Rd = 0xFFFFFFFF
- Rs1 = 0x80000000, Rs2 = 0x7FFFFFFF, Rd = 0x00000000
- Rs1 = 0x80000000, Rs2 = 0x40000000, Rd = 0x20000000
- Rs1 = 0x80000001, Rs2 = 0x00000001, Rd = 0x40000000

# Chapter 9. New User Control & Status Registers

## Brief Summary

Symbolic Mnemonics	CSR Address				Hex	Privilege
	[11:10]	[9:8]	[7:6]	[5:0]		
vxsat	00	00	00	001001	0x009	URW

## 9.1. Fixed-point Saturation Flag Register

**Mnemonic Name:** vxsat

**IM Requirement:** misa.P == 1

### Access Rules:

This specification does not require that the mstatus.VS field be implemented. However, if the mstatus.VS field *is* implemented (e.g., because the V extension is implemented), then instructions in this specification that access the vxsat register raise illegal-instruction exceptions when mstatus.VS=Off. Furthermore, if mstatus.VS=Initial or Clean, instructions in this specification that write vxsat cause mstatus.VS to be set to Dirty.

Similarly, this specification does not require that the vsstatus.VS field be implemented. However, if the vsstatus.VS field *is* implemented and the current virtualization mode, V, is 1, the same rules additionally apply to vsstatus.VS field.

**Access Mode:** User

**CSR Address:** 0x009 (standard read/write)

**XLEN:** 64 and 32

This register stores the overflow/saturation 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 P extension instructions when a saturated result is generated.	RW	0
Reserved	[XLEN-1:1]	Reserved	RAZWI	0

# Chapter 10. Instruction Encoding Table

Table 36. P Extension Instruction Encoding for funct3[14:12]==0b000.

funct3 == 000								
funct7	[2:0]							
[6:3]	000	001	010	011	100	101	110	111
0000	PAADD.H (radd16)	PASUB.H (rsub16)	PAAS.HX (rcras16)	PASA.HX (rcrsa16)	PAADD.B (radd8)	PASUB.B (rsub8)	PMSLT.H (scmplt16)	PMSLT.B (scmplt8)
0001	PSADD.H (kadd16)	PSSUB.H (ksub16)	PSAS.HX (kcras16)	PSSA.HX (kcsra16)	PSADD.B (kadd8)	PSSUB.B (ksub8)	PMSGE.H (scmple16)	PMSGE.B (scmple8)
0010	PAADDU.H (uradd16)	PASUBU.H (ursub16)	urcras16	urcrsa16	PAADDU.B (uradd8)	PASUBU.B (ursub8)	PMSLTU.H (ucmplt16)	PMSLTU.B (ucmplt8)
0011	PSADDU.H (ukadd16)	PSSUBU.H (uksub16)	ukcras16	ukcsra16	PSADDU.B (ukadd8)	PSSUBU.B (uksub8)	PMSGEU.H (ucmple16)	PMSGEU.B (ucmple8)
0100	PADD.H (add16)	PSUB.H (sub16)	PAS.HX (cras16)	PSA.HX (crsa16)	PADD.B (add8)	PSUB.B (sub8)	PMSEQ.H (cmpeq16)	PMSEQ.B (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/kslli 16		srai8/.u	srl18/.u	slli8/kslli8	
1000	PMIN.H (smin16)	PMAX.H (smax16)	sclip16/uc lip16	PMULQ.H (khw16)	PMIN.B (smin8)	PMAX.B (smax8)	sclip8/ucli p8	khm8
1001	PMINU.H (umin16)	PMAXU.H (umax16)		khmx16	PSMINU.B (umin8)	PMAXU.B (umax8)		khmx8
1010	smul16	smulx16			smul8	smulx8	oneop	oneop2
1011	umul16	umulx16			umul8	umulx8		
1100					smaqa	smaqa.su	umaqa	wext
1101								wexti
1110	ave		sclip32	bitrev	bitrevi (RV32/RV 64)	bitrevi (RV64)		
1111			uclip32				PDIFSUMU.B (pbsad)	PDIFSUMUA.B (pbsada)

Table 37. Instruction Encoding for funct3[14:12]==0b000 and funct7[31:25]==0b1010110 (oneop).

ONEOP == 1010110								
subf5	[2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	insb (RV32/RV64)				insb (RV64)			
01	sunpkd81 0	sunpkd82 0	sunpkd83 0	sunpkd83 1	zunpkd81 0	zunpkd82 0	zunpkd83 0	zunpkd83 1
10	kabs8	PABS.H (kabs16)	PABS.W (kabs32)	sunpkd83 2	ABS (kabsw)			zunpkd83 2
11	swap8							

Table 38. Instruction Encoding for  $\text{funct3}[14:12] == 0b000$  and  $\text{funct7}[31:25] == 0b1010111$  (oneop2).

ONEOP2 == 1010111								
subf5	[2:0]							
[4:3]	000	001	011	010	100	101	111	110
00	clrs8	clz8						
01	clrs16	clz16						
11	clrs32	clz32						
10								

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	SADD (kaddw)	SSUB (ksubw)	kaddh	ksubh	PMUL.W.HEE (smbb16)	kdmbb	khmbb	pkbb16 (RV64)
0001	SADDU (ukaddw)	SSUBU (uksubw)	ukaddh	uksubh	PMUL.W.HEO (smbt16)	kdmbt	khmbt	pktb16
0010	AADD (raddw)	ASUB (rsubw)	sra.u	ksllw	PMULW.HOO (smtt16)	kdmtt	khmtt	pktt16 (RV64)
0011	AADDU (uraddw)	ASUBU (ursubw)	sraiw.u	kslliw	kmada	kmxda		pktb16
0100	PMULH.W (smmul)	kmmsb	PMUH.W.HE (smmwb)	PMHACC.W.HE (kmmawb)	kmada	kmaxda	kmsda	kmsxda
0101	MULHR PMULHR.W (smmul.u)	kmmsb.u	smmwbu	kmmawb.u	smds	PMACC.W.HEE (kmabb)	kmads	smal
0110	MHACC PMHACC.W (kmmac)	MULQ PMULQ.W (kwmmul)	PMULH.W.HO (smmwt)	PMHACC.W.O (kmmawt)	smdrs	PMACC.W.HEO (kmabt)	kmadrs	kslraw
0111	MHRACC PMHRACC.W (kmmac.u)	MULQR PMULQR.W (kwmmul.u)	smmwu	kmmawt.u	smxds	PMAC.W.HOO (kmatt)	kmaxds	kslraw.u
1000	radd64	rsub64	PM2ADDA.W (smar64)	smsr64	smalbb	smalds	PM4ADDA.H (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	ADDD (add64)	SUBD (sub64)	maddr32	msubr32				kmmawb 2
1101		kdmbb	srai.u (RV32/RV 64)	srai.u (RV64)	kdmbb1 6	kdmbb16	khmbb16	kmmawb 2.u
1110	mulsr64	kdmbt			kdmbt16	kdmbt16	khmbt16	kmmawt 2
1111	mulr64	kdmat			kdmat16	kdmat16	khmat16	kmmawt 2.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	PAADD.W (radd32)	PASUB.W (rsub32)	PAAS.WX (rcras32)	PASA.WX (rcrsa32)				
0001	PSADD.W (kadd32)	PSSUB.W (ksub32)	PSAS.WX (kcras32)	PSSA.WX (kcrsa32)	MUL.WEO (smbt32)			pkbt32
0010	PAADDU.W (uradd32)	PASUBU.W (ursub32)	urcras32	urcrsa32	MUL.WOO (smmt32)			
0011	PSADDU.W (ukadd32)	PSSUBU.W (uksub32)	ukcras32	ukcrsa32	PM2ADD.W (kmada32)	PM2ADD.WX (kmxda32)		pktb32
0100	PADD.W (add32)	PSUB.W (sub32)	PAS.WX (cras32)	PSA.WX (crsa32)		PM2ADD.WX (kmada32)	kmsda32	kmsxda3 2
0101	sra32	srl32	sll32	kslra32	smds32	MACC.WEE (kmabb32)	kmads32	
0110	sra32.u	srl32.u	ksll32	kslra32.u	PM2SUB.W (smdrs32)	MACC.WEO (kmabt32)	PM2SUB.W (kmadr32)	
0111	srai32	srl32	slli32		PM2SUB.WX (smxds32)	MACC.WOO (kmatt32)	PM2SUBA.W ((Kmaxds 32))	
1000	srai32.u	srl32.u	kslli32					
1001	PMIN.W (smin32)	PMAX.W (smax32)						
1010	PMINU.W (umin32)	PMAXU.W (umax32)						
1011	rstas32	rstsa32	rstas16	rstsa16				
1100	kstas32	kstsa32	kstas16	kstsa16				
1101	urstas32	urstsa32	urstas16	urstsa16				
1110	ukstas32	ukstsa32	ukstas16	ukstsa16				
1111	stas32	sts32	stas16	sts32				

Table 41. P Extension Instruction Encoding for  $\text{funct3}[14:12] == 0b011$ .

<b>funct3 = 011</b>								
<b>funct7</b>	<b>[2:0]</b>							
<b>[6:3]</b>	000	001	010	011	100	101	110	111
<b>0000</b>								
<b>0001</b>								
<b>0010</b>								
<b>0011</b>								
<b>0100</b>								
<b>0101</b>								
<b>0110</b>								
<b>0111</b>								
<b>1000</b>								
<b>1001</b>								
<b>1010</b>								
<b>1011</b>								
<b>1100</b>								
<b>1101</b>								
<b>1110</b>								
<b>1111</b>								

# **Chapter 11. Removed Instructions Due to RVB overlaps (hidden for now)**

# **Appendix A: Instruction Latency and Throughput (hidden for now)**

# Appendix B: instructions from original draft that are still up for discussion and renaming.

## 1. ADDD (ADD64) (64-bit Addition)

Type: RV32 Only

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ADDD 1100000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
ADDD 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  $2d+1$ .

Exceptions: None

Privilege level: All

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

Intrinsic functions:

```
int64_t _rv_sadd64(int64_t a, int64_t b);  
uint64_t _rv_uadd64(uint64_t a, uint64_t b);
```

## 2. 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	OP-P 1110111

Syntax:

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

Purpose: Calculate the average of the contents of two general purpose 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:

RV32:

```
res33 = SE33(Rs1) + SE33(Rs2) + SE33(1);  
Rd = res33[32:1];
```

RV64:

```
res65 = SE65(Rs1) + SE65(Rs2) + SE65(1);  
Rd = res65[64:1];
```

Exceptions: None

Privilege level: All

Note:

### 3. BITREV (Bit Reverse)

Type: DSP

Replaced with REV in Zbpbo extension.

Format:

31    25	24    20	19    15	14    12	11    7	6    0
BITREV 1110011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

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

Purpose: Reverse the bits 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 bits 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]; // RV32
msb = Rs2[5:0]; // RV64
rev[0:msb] = Rs1[msb:0];
Rd = ZE32(rev[msb:0]); // RV32 Rd
= ZE64(rev[msb:0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uintXLEN_t __rv_bitrev(uintXLEN_t a, uint32_t msb);
```

## 4. BITREVI (Bit Reverse Immediate)

Type: DSP

Replaced with REV in Zbpbo extension.

Format:

RV32

31 25	24 20	19 15	14 12	11 7	6 0
BITREVI 1110100	imm5u	Rs1	000	Rd	OP-P 1110111

RV64

31 26	25 20	19 15	14 12	11 7	6 0
BITREVI 111010	imm6u	Rs1	000	Rd	OP-P 1110111

Syntax:

```
BITREVI Rd, Rs1, imm5u (RV32)
BITREVI Rd, Rs1, imm6u (RV64)
```

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

**Description:** This instruction reverses the bits of the content of Rs1. The reversed bit width is calculated as imm5u+1 (RV32) or imm6u+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 = imm5u; // RV32
msb = imm6u; // RV64
rev[0:msb] = Rs1[msb:0];
Rd = ZE32(rev[msb:0]);    // RV32
Rd = ZE64(rev[msb:0]);    // RV64
```

Exceptions: None

Privilege level: All

Note:

### Intrinsic functions:

The intrinsic function of this instruction is the same as the intrinsic function of "BITREV" instruction. A compiler can detect constant value in the function msb argument and use this instruction.

```
uintXLEN_t __rv_bitrev(uintXLEN_t a, uint32_t msb);
```

## 5. CLROV (Clear OV flag)

Type: DSP

Format:

31 20	19 15	14 12	11 7	6 0
vxsat (0x009) 000000001001	00001	111	Rd	SYSTEM 1110011

Syntax:

```
CLROV # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias for "CSRRCI x0, vxsat, 1" instruction.

Intrinsic functions:

```
void __rv_clrov(void);
```

## 6. 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	OP-P 1110111

Syntax:

**CLRS8 Rd, Rs1**

**Purpose:** Count the number of redundant sign bits of the 8-bit elements of a general purpose 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:

- Required:

```
uintXLEN_t __rv_clrs8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t __rv_v_clrs8(int8x4_t a);
```

RV64:

```
uint8x8_t __rv_v_clrs8(int8x8_t a);
```

## 7. 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	OP-P 1110111

Syntax:

```
CLRS16 Rd, Rs1
```

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

- Required:

```
uintXLEN_t __rv_clrs16(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_clrs16(int16x2_t a);
```

RV64:

```
uint16x4_t __rv_v_clrs16(int16x4_t a);
```

## 8. 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	OP-P 1110111

Syntax:

```
CLRS32 Rd, Rs1
```

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

- Required:

```
uintXLEN_t _rv_clrs32(intXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint32x2_t _rv_v_clrs32(int32x2_t a);
```

## 9. 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	OP-P 1110111

Syntax:

**CLZ8 Rd, Rs1**

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

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t _rv_v_clz8(uint8x4_t a);
```

RV64:

```
uint8x8_t _rv_v_clz8(uint8x8_t a);
```

## 10. 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	OP-P 1110111

Syntax:

CLZ16 Rd, Rs1

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

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_clz16(uint16x2_t a);
```

RV64:

```
uint16x4_t _rv_v_clz16(uint16x4_t a);
```

## 11. CLZ32 (SIMD 32-bit Count Leading Zero)

Type: SIMD

RV32: Replaced with CLZ in Zbpbo. RV64: Zpn

Format:

31 25	24 20	19 15	14 12	11 7	6 0
ONEOP2 1010111	CLZ32 11001	Rs1	000	Rd	OP-P 1110111

Syntax:

CLZ32 Rd, Rs1

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

- Required:

```
uintXLEN_t __rv_clz32(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint32x2_t __rv_v_clz32(uint32x2_t a);
```

## 12. INSB (Insert Byte)

Type: DSP

Format:

RV32

31	25	24	23	22	21	20	19	15	14	12	11	7	6	0
ONEOP 1010110		INSB 00		0		imm2u		Rs1		000		Rd		OP-P 1110111

RV64

31	25	24	23	22	20	19	15	14	12	11	7	6	0
ONEOP 1010110		INSB 00			imm3u		Rs1		000		Rd		OP-P 1110111

Syntax:

(RV32) INSB Rd, Rs1, imm2u

(RV64) INSB Rd, Rs1, imm3u

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 "imm2u" (RV32) or "imm3u" (RV64) of Rd.

Operations:

```
bpos = imm2u; // RV32
bpos = imm3u; // RV64
Rd.B[bpos] = Rs1.B[0]
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uintXLEN_t _rv_insb(uintXLEN_t t, uintXLEN_t a, uint32_t bpos);
```

## 13. 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	OP-P 1110111

Syntax:

```
KABS8 Rd, Rs1
```

Purpose: Compute the absolute value of 8-bit signed integer elements in parallel.

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:

- Required:

RV32:

```
int8x4_t __rv_v_kabs8(int8x4_t a);
```

RV64:

```
int8x8_t __rv_v_kabs8(int8x8_t a);
```

## 14. KADD64 (64-bit Signed Saturating Addition)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
KADD64 1001000	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**RV64 Description:** This instruction adds the 64-bit signed integer in Rs1 with the 64-bit signed integer in Rs2. If the result exceeds 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);
a65 = SE65(R[a_H].R[a_L]);
b65 = SE65(R[b_H].R[b_L]);
res65 = a65 + b65;
if (res65 > (2^63)-1) {
    res65 = (2^63)-1; OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63; OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

RV64:

```
a65 = SE65(Rs1);
b65 = SE65(Rs2);
res65 = a65 + b65;
if (res65 > (2^63)-1) {
    res65 = (2^63)-1; OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63; OV = 1;
}
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int64_t __rv_kadd64(int64_t a, int64_t b);
```

## 15. 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	OP-P 1110111

Syntax:

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

Purpose: Add the signed lower 16-bit content of two registers with Q15 saturation.

Description: The signed lower 16-bit content of Rs1 is added with the signed lower 16-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:

```
a17 = SE17(Rs1.H[0]);  
b17 = SE17(Rs2.H[0]);  
res17 = a17 + b17;  
if (res17 > (2^15)-1) {  
    res17 = (2^15)-1;  
    OV = 1;  
} else (res17 < -2^15) {  
    res17 = -2^15;  
    OV = 1  
}  
Rd = SE32(res17.H[0]); // RV32  
Rd = SE64(res17.H[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int32_t _rv_kaddh(int16_t a, int16_t b);
```

## 16. KDMBB, KDMBT, KDMTT

KDMBB (Signed Saturating Double Multiply B16 x B16)

## KDMBT (Signed Saturating Double Multiply B16 x T16)

## 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	OP-P 1110111

KDMBT

31 25	24 20	19 15	14 12	11 7	6 0
KDMBT 0001101	Rs2	Rs1	001	Rd	OP-P 1110111

KDMTT

31 25	24 20	19 15	14 12	11 7	6 0
KDMTT 0010101	Rs2	Rs1	001	Rd	OP-P 1110111

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

```
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)) {
    resQ30[31:0] = aop s* bop;
    shifted[31:0] = resQ30[31:0] << 1;
    resQ31 = shifted[31:0];
    Rd = resQ31;           // RV32
    Rd = SE64(resQ31); // RV64
} else {
    resQ31 = 0xFFFFFFFF;
    Rd = resQ31;           // RV32
    Rd = SE64(resQ31); // RV64
    OV = 1;
}

```

**Exceptions:** None

**Privilege level:** All

**Note:**

**Intrinsic functions:**

- KDMBB
- Required:

```
int32_t __rv_kdmbb(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int32_t __rv_v_kdmbb(int16x2_t a, int16x2_t b);
RV64:
int32_t __rv_v_kdmbb(int16x2_t a, int16x2_t b);

```

- KDMBT
- Required:

```
int32_t __rv_kdmbt(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int32_t __rv_v_kdmbt(int16x2_t a, int16x2_t b);
RV64:
int32_t __rv_v_kdmbt(int16x2_t a, int16x2_t b);

```

- KDMTT

- Required:

```
int32_t __rv_kdmmt(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kdmmt(int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_kdmmt(int16x2_t a, int16x2_t b);
```

## 17. KDMABB, KDMABT, KDMATT

**KDMABB (Signed Saturating Double Multiply Addition B16 x B16)**

**KDMABT (Signed Saturating Double Multiply Addition B16 x T16)**

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	OP-P 1110111

**KDMABT**

31    25	24    20	19    15	14    12	11    7	6    0
KDMABT 1110001	Rs2	Rs1	001	Rd	OP-P 1110111

**KDMATT**

31    25	24    20	19    15	14    12	11    7	6    0
KDMATT 1111001	Rs2	Rs1	001	Rd	OP-P 1110111

**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 signed lower 32-bit word of the destination register and write the signed 32-bit 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 signed lower 32-bit word of the destination register and write the signed 32-bit saturated addition result into the destination register.

the **int32\_t \_rv\_kdmabb(int32\_t t, uint32\_t a, uint32\_t b);**  
writ

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)) {  
    resQ30[31:0] = aop s* bop;  
    shifted[31:0] = resQ30[31:0] << 1;  
    resQ31 = shifted[31:0];  
} else {  
    resQ31 = 0x7FFFFFFF;  
    OV = 1;  
}  
c33 = SE33(Rd.W[0]);  
d33 = SE33(resQ31);  
tmp33 = c33 + d33;  
if (tmp33 s> (2^31)-1) {  
    resadd32 = (2^31)-1;  
    OV = 1;  
} else if (tmp33 s< -2^31) {  
    resadd32 = -2^31;  
    OV = 1;  
} else {  
    resadd32 = tmp33.W[0];  
}  
Rd = resadd32;      // RV32  
Rd = SE64(resadd32); // RV64
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KDMABB
- Required:

RV32:

```
int32_t _rv_v_kdmabb(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_kdmabb(int32_t t, int16x2_t a, int16x2_t b);
```

- KDMABT

- Required:

```
int32_t __rv_kdmabt(int32_t t, uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_kdmabt(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_kdmabt(int32_t t, int16x2_t a, int16x2_t b);
```

- KDMATT

- Required:

```
int32_t __rv_kdmatt(int32_t t, uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_kdmatt(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_kdmatt(int32_t t, int16x2_t a, int16x2_t b);
```

## 18. KHM8, KHMX8

KHM8 (SIMD Signed Saturating Q7 Multiply)

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	OP-P 1110111

KHMX8

31 25	24 20	19 15	14 12	11 7	6 0
KHMX8 1001111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

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

Purpose: Perform Q7xQ7 element multiplications in parallel. 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 "KHMX8" 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 "KHX8") {
    op1t = Rs1.B[x+1]; op2t = Rs2.B[x];     // Rs1 top
    op1b = Rs1.B[x];   op2b = Rs2.B[x+1]; // Rs1 bottom
}
for ((aop,bop,res16) in [(op1t,op2t,rest), (op1b,op2b,resb)]) {
    if ((0x80 != aop) || (0x80 != bop)) {
        mres[15:0] = aop s* bop;
        rshifted[15:0] = mres[15:0] s>> 7;
        res16 = rshifted[15:0];
    } else {
        res16= 0x007F;
        OV = 1;
    }
}
Rd.H[x/2] = concat(rest.B[0], resb.B[0]);
```

```
for RV32, x=0,2
for RV64, x=0,2,4,6
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KHM8
- Required:

```
uintXLEN_t __rv_khm8(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t __rv_v_khm8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __rv_v_khm8(int8x8_t a, int8x8_t b);
```

- KHMX8
- Required:

```
uintXLEN_t __rv_khmx8(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t __rv_v_khmx8(int8x4_t a, int8x4_t b);
```

RV64:

```
int8x8_t __rv_v_khmx8(int8x8_t a, int8x8_t b);
```

## 19. KHMBB, KHMBT, KHMTT

**KHMBB (Signed Saturating Half Multiply B16 x B16)**

**KHMBT (Signed Saturating Half Multiply B16 x T16)**

**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	OP-P 1110111

**KHMBT**

31    25	24    20	19    15	14    12	11    7	6    0
KHMBT 0001110	Rs2	Rs1	001	Rd	OP-P 1110111

**KHMTT**

31    25	24    20	19    15	14    12	11    7	6    0
KHMTT 0010110	Rs2	Rs1	001	Rd	OP-P 1110111

**Syntax:**

**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]; // KHMBB
aop = Rs1.H[0]; bop = Rs2.H[1]; // KHMBT
aop = Rs1.H[1]; bop = Rs2.H[1]; // KHMTT
```

```
If ((0x8000 != aop) || (0x8000 != bop)) {
    res[31:0] = aop s* bop;
    rshifted[31:0] = res[31:0] s>> 15;
    Mres32 = rshifted[31:0];
} else {
    Mres32 = 0x00007FFF;
    OV = 1;
}
Rd = SE32(Mres32.H[0]); // Rv32
Rd = SE64(Mres32.H[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- **KHMBB**

- Required:

```
int32_t __rv_khmbb(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_khmbb(int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_khmbb(int16x2_t a, int16x2_t b);
```

- **KHMKT**

- Required:

```
int32_t __rv_khmkt(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_khmkt(int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_khmkt(int16x2_t a, int16x2_t b);
```

- **KHMTT**

- Required:

```
int32_t __rv_khmtt(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_khmtt(int16x2_t a, int16x2_t b);
```

RV64:

```
int32_t __rv_v_khmtt(int16x2_t a, int16x2_t b);
```

## 20. KMADA, KMAXDA

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

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	OP-P 1110111

KMAXDA

31    25	24    20	19    15	14    12	11    7	6    0
KMAXDA 0100101	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

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

**Purpose:** Perform 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 exceeds 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:

```
mula32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[1]; // KMADA  
mulb32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[0]; //  
mula32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[0]; // KMAXDA  
mulb32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[1]; //
```

```
res34[x] = SE34(Rd.W[x]) + SE34(mula32[x]) + SE34(mulb32[x]);  
if (res34[x] > (2^31)-1) {  
    res34[x] = (2^31)-1;  
    OV = 1;  
} else if (res34[x] < -2^31) {  
    res34[x] = -2^31;  
    OV = 1;  
}  
Rd.W[x] = res34[x].W[0];  
for RV32: x=0  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KMADA
- Required:

```
intXLEN_t __rv_kmada(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmada(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

- KMAXDA
- Required:

```
intXLEN_t __rv_kmaxda(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmaxda(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

## 21. KMADS, KMADRS, KMAXDS

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

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

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	OP-P 1110111

KMADRS

31 25	24 20	19 15	14 12	11 7	6 0
KMADRS 0110110	Rs2	Rs1	001	Rd	OP-P 1110111

KMAXDS

31 25	24 20	19 15	14 12	11 7	6 0
KMAXDS 0111110	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

KMADS Rd, Rs1, Rs2

KMADRS Rd, Rs1, Rs2

KMAXDS Rd, Rs1, Rs2

Purpose: Perform 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 exceeds 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:

```
mula32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[1]; // KMADS
mulb32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[0]; //
mula32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[0]; // KMADRS
mulb32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[1]; //
mula32[x] = Rs1.W[x].H[1] s* Rs2.W[x].H[0]; // KMAXDS
mulb32[x] = Rs1.W[x].H[0] s* Rs2.W[x].H[1]; //
```

```
res34[x] = SE34(Rd.W[x]) + SE34(mula32[x]) - SE34(mulb32[x]);
if (res34[x] > (2^31)-1) {
    res34[x] = (2^31)-1;
    OV = 1;
} else if (res34[x] < -2^31) {
    res34[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res34[x];
for RV32: x=0
for RV64: x=1..0
    uintXLEN_t __rv_kabs8(uintXLEN_t a);
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KMADS
- Required:

```
intXLEN_t __rv_kmads(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmads(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

- KMADRS

- Required:

```
intXLEN_t __rv_kmadrs(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmadrs(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

- KMAXDS

- Required:

```
intXLEN_t __rv_kmaxds(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmaxds(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

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

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31      25	24      20	19      15	14      12	11      7	6      0
KMAR64 1001010	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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);
mul65 = SE65(Rs1 s* Rs2);
top65 = SE65(R[t_H].R[t_L]);
res65 = top65 + mul65;
if (res65 > (2^63)-1) {
    res65 = (2^63)-1;
    OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63;
    OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

RV64:

```
mula66 = SE66(Rs1.W[0] s* Rs2.W[0]);
mulb66 = SE66(Rs1.W[1] s* Rs2.W[1]);
res66 = SE66(Rd) + mula66 + mulb66;
if (res66 > (2^63)-1) {
    res66 = (2^63)-1;
    OV = 1;
} else if (res66 < -2^63) {
    res66 = -2^63;
    OV = 1;
}
Rd = res66.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t _rv_kmar64(int64_t t, intXLEN_t a, intXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
int64_t _rv_v_kmar64(int64_t t, int32x2_t a, int32x2_t b);
```

## 23. KMDA, KMXDA

### KMDA (SIMD Signed Multiply Two Halfs and Add)

**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	OP-P 1110111

**KMXDA**

31 25	24 20	19 15	14 12	11 7	6 0
KMXDA 0011101	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

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

**Purpose:** Perform 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] s* Rs2.W[x].H[1]) + (Rs1.W[x].H[0] s* Rs2.W[x].H[0]);  
    // KMXDA  
    Rd.W[x] = (Rs1.W[x].H[1] s* Rs2.W[x].H[0]) + (Rs1.W[x].H[0] s* 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

- Required:

```
intXLEN_t __rv_kmda(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmda(int16x2_t a, int16x2_t b);
```

RV64:

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

- KMXDA

- Required:

```
intXLEN_t __rv_kmxda(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmxda(int16x2_t a, int16x2_t b);
```

RV64:

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

```
uintXLEN_t __rv_kabs8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

## 24. KMMAWB2, KMMAWB2.u

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

**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	OP-P 1110111

**KMMAWB2.u**

31    25	24    20	19    15	14    12	11    7	6    0
KMMAWB2.u 1101111	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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) or (Rs2.W[x].H[0] != 0x8000)) {
    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
} else {
    addop.W[x] = 0x7fffffff;
    OV = 1;
}
res33[x] = SE33(Rd.W[x]) + SE33(addop.W[x]);
if (res33[x] s> (2^31)-1) {
    res33[x] = (2^31)-1;
    OV = 1;
} else if (res33[x] s< -2^31) {
    res33[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res33[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- **KMMAWB2**
- Required:

```
intXLEN_t __rv_kmmawb2(intXLEN_t t, intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmawb2(int32_t t, int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmawb2(int32x2_t t, int32x2_t a, int16x4_t b);
```

—

```
intXLEN_t __rv_kmmawb2_u(intXLEN_t t, intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmawb2_u(int32_t t, int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmawb2_u(int32x2_t t, int32x2_t a, int16x4_t b);
```

## 25. KMMAWT2, KMMAWT2.u

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

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	OP-P 1110111

KMMAWT2.u

31 25	24 20	19 15	14 12	11 7	6 0
KMMAWT2.u 1111111	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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] = 0xffffffff;  
    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  
}  
res33[x] = SE33(Rd.W[x]) + SE33(addop.W[x]);  
if (res33[x] s> (2^31)-1) {  
    res33[x] = (2^31)-1;  
    OV = 1;  
} else if (res33[x] s< -2^31) {  
    res33[x] = -2^31;  
    OV = 1;  
}  
Rd.W[x] = res33[x].W[0];  
for RV32: x=0  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KMMAWT2
- Required:

```
intXLEN_t __rv_kmmawt2(intXLEN_t t, intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmawt2(int32_t t, int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmawt2(int32x2_t t, int32x2_t a, int16x4_t b);
```

- KMMAWT2.u

- Required:

```
intXLEN_t __rv_kmmawt2_u(intXLEN_t t, intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmawt2_u(int32_t t, int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmawt2_u(int32x2_t t, int32x2_t a, int16x4_t b);
```

## 26. KMMSB, KMMSB.u

KMMSB (SIMD Saturating MSW Signed Multiply Word and Subtract)

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	OP-P 1110111

KMMSB.u

31      25	24      20	19      15	14      12	11      7	6      0
KMMSB.u 0101001	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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;
    res33[x] = SE33(Rd.W[x]) - SE33(Round[x][32:1]);
} else {
    res33[x] = SE33(Rd.W[x]) - SE33(Mres[x][63:32]);
}
if (res33[x] > (2^31)-1) {
    res33[x] = (2^31)-1;
    OV = 1;
} else if (res33[x] < -2^31) {
    res33[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res33[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KMMSB
- Required:

```
intXLEN_t __rv_kmmsb(intXLEN_t t, intXLEN_t a, intXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

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

- KMMSB.u
- Required:

```
intXLEN_t __rv_kmmsb_u(intXLEN_t t, intXLEN_t a, intXLEN_t b);
```

RV64:

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

## 27. KMMWB2, KMMWB2.u

### KMMWB2 (SIMD Saturating MSW Signed Multiply Word and Bottom Half & 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	OP-P 1110111

KMMWB2.u

31    25	24    20	19    15	14    12	11    7	6    0
KMMWB2.u 1001111	Rs2	Rs1	001	Rd	OP-P 1110111

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] = 0xffffffff;  
    OV = 1;  
} else {  
    Mres[x][47:0] = Rs1.W[x] s* Rs2.W[x].H[0];  
    if (“.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
- Required:

```
intXLEN_t __rv_kmmwb2(intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

```
RV32:  
    int32_t __rv_v_kmmwb2(int32_t a, int16x2_t b);  
RV64:  
    int32x2_t __rv_v_kmmwb2(int32x2_t a, int16x4_t b);
```

- KMMWB2.u
- Required:

```
intXLEN_t __rv_kmmwb2_u(intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmwb2_u(int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmwb2_u(int32x2_t a, int16x4_t b);
```

## 28. KMMWT2, KMMWT2.u

KMMWT2 (SIMD Saturating MSW Signed Multiply Word and Top Half & 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	OP-P 1110111

KMMWT2.u

31    25	24    20	19    15	14    12	11    7	6    0
KMMWT2.u 1011111	Rs2	Rs1	001	Rd	OP-P 1110111

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 (“.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
- Required:

```
intXLEN_t __rv_kmmwt2(intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmmwt2(int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmwt2(int32x2_t a, int16x4_t b);
```

- KMMWT2.u
- Required:

```
intXLEN_t __rv_kmmwt2_u(intXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_kmmwt2_u(int32_t a, int16x2_t b);
```

RV64:

```
int32x2_t __rv_v_kmmwt2_u(int32x2_t a, int16x4_t b);
```

## 29. KMSDA, KMSXDA

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

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	OP-P 1110111

KMSXDA

31 25	24 20	19 15	14 12	11 7	6 0
KMSXDA 0100111	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

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

Purpose: Perform 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.

bit elements of Rd. If the subtraction result exceeds 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:

```
mula34[x] = SE34(Rs1.W[x].H[1] s* Rs2.W[x].H[1]); // KMSDA
mulb34[x] = SE34(Rs1.W[x].H[0] s* Rs2.W[x].H[0]); //
mula34[x] = SE34(Rs1.W[x].H[1] s* Rs2.W[x].H[0]); // KMSXDA
mulb34[x] = SE34(Rs1.W[x].H[0] s* Rs2.W[x].H[1]); //
```

```
res34[x] = SE34(Rd.W[x]) - mula34[x] - mulb34[x];
if (res34[x] > (2^31)-1) {
    res34[x] = (2^31)-1;
    OV = 1;
} else if (res34[x] < -2^31) {
    res34[x] = -2^31;
    OV = 1;
}
Rd.W[x] = res34[x].W[0];
for RV32: x=0
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- KMSDA
- Required:

```
intXLEN_t __rv_kmsda(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_kmsda(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

- KMSXDA
- Required:

```
intXLEN_t __rv_kmsxda(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_kmsxda(int32_t t, int16x2_t a, int16x2_t b);
```

RV64:

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

```
uintXLEN_t _rv_kabs8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

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

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
KMSR64 1001011	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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);
mul65 = SE65(Rs1 s* Rs2);
top65 = SE65(R[t_H].R[t_L]);
res65 = top65 - mul65;
if (res65 > (2^63)-1) {
    res65 = (2^63)-1;
    OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63;
    OV = 1;
}
R[t_H].R[t_L] = res65.D[0];

```

RV64:

```

mula66 = SE66(Rs1.W[0] s* Rs2.W[0]);
mulb66 = SE66(Rs1.W[1] s* Rs2.W[1]);
res66 = SE66(Rd) - mula66 - mulb66;
if (res66 > (2^63)-1) {
    res66 = (2^63)-1;
    OV = 1;
} else if (res66 < -2^63) {
    res66 = -2^63;
    OV = 1;
}
Rd = res66.D[0];

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_kmsr64(int64_t t, intXLEN_t a, intXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
int64_t __rv_v_kmsr64(int64_t t, int32x2_t a, int32x2_t b);
```

- Optional (e.g., GCC vector extensions):

# 31. 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	OP-P 1110111

Syntax:

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

**Purpose:** Perform 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 = 0xffffffff; OV = 1;
} else if (res < -2^31) {
    res = 0x80000000; OV = 1;
}
Rd = res.W[0]; // RV32
Rd = SE64(res.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int32_t __rv_ksllw(int32_t a, uint32_t b);
```

## 32. 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	OP-P 1110111

Syntax:

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

Purpose: Perform 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 = 0xffffffff; OV = 1;
} else if (res < -2^31) {
    res = 0x80000000; OV = 1;
}
Rd = res.W[0];      // RV32
Rd = SE64(res.W[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
#include <rvkabs.h>
uintXLEN_t _rv_kabs8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

### 33. 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	OP-P 1110111

Syntax:

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

Purpose: Perform 8-bit elements logical left shift operations with saturation in parallel. 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.B[0];
} else {
    Rd = Rs1;
}
for RV32: x=3..0,
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t __rv_ksll8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t __rv_v_ksll8(int8x4_t a, uint32_t b);
```

RV64:

```
int8x8_t __rv_v_ksll8(int8x8_t a, uint32_t b);
```

## 34. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

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

Purpose: Perform 8-bit elements logical left shift operations with saturation in parallel. 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;
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.B[0];
} else {
    Rd = Rs1;
}
for RV32: x=3..0,
for RV64: x=7..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t __rv_ksll8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t __rv_v_ksll8(int8x4_t a, uint32_t b);
```

RV64:

```
int8x8_t __rv_v_ksll8(int8x8_t a, uint32_t b);
```

## 35. 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		OP-P 1110111	

Syntax:

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

Purpose: Perform 16-bit elements logical left shift operations with saturation in parallel. 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.H[0];
} else {
    Rd = Rs1;
}
for RV32: x=1..0,
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t _rv_ksll16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t _rv_v_ksll16(int16x2_t a, uint32_t b);
```

RV64:

```
int16x4_t _rv_v_ksll16(int16x4_t a, uint32_t b);
```

## 36. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

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

Purpose: Perform 16-bit elements logical left shift operations with saturation in parallel. 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;
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.H[0];
} else {
    Rd = Rs1;
}
for RV32: x=1..0,
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t __rv_ksll16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_ksll16(int16x2_t a, uint32_t b);
```

RV64:

```
int16x4_t __rv_v_ksll16(int16x4_t a, uint32_t b);
```

## 37. KSLRA8, KSLRA8.u

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

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	OP-P 1110111

KSLRA8.u

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA8.u 0110111	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

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

**Purpose:** Perform 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]==-2<sup>3</sup> (0x8)" is defined to be equivalent to the behavior of "Rs2[3:0]==-(2<sup>3</sup>-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] u<< 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
- Required:

```
uintXLEN_t __rv_kslra8(uintXLEN_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int8x4_t __rv_v_kslra8(int8x4_t a, int32_t b);
RV64:
int8x8_t __rv_v_kslra8(int8x8_t a, int32_t b);

```

```
uintXLEN_t __rv_kslra8_u(uintXLEN_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t _rv_v_kslra8_u(int8x4_t a, int32_t b);
```

RV64:

```
int8x8_t _rv_v_kslra8_u(int8x8_t a, int32_t b);
```

## 38. KSLRA16, KSLRA16.u

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

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	OP-P 1110111

KSLRA16.u

31 25	24 20	19 15	14 12	11 7	6 0
KSLRA16.u 0110011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

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

**Purpose:** Perform 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]==-2<sup>4</sup> (0x10)" is defined to be equivalent to the behavior of "Rs2[4:0]==-(2<sup>4</sup>-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 result.

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] u<< 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
- Required:

```
uintXLEN_t __rv_kslra16(uintXLEN_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_kslra16(int16x2_t a, int32_t b);
```

RV64:

```
int16x4_t __rv_v_kslra16(int16x4_t a, int32_t b);
```

- KSLRA16.u

- Required:

```
uintXLEN_t __rv_kslra16_u(uintXLEN_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_kslra16_u(int16x2_t a, int32_t b);
```

RV64:

```
int16x4_t __rv_v_kslra16_u(int16x4_t a, int32_t b);
```

## 39. 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	OP-P 1110111

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] s>> sa;
} else {
    sa = Rs2[5:0];
    tmp[(31+sa):0] = Rs1.W[0] u<< 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:**

```
intXLEN_t __rv_kslraw(int32_t a, int32_t b);
```

## 40. 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	OP-P 1110111

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[(31+sa):0] = Rs1.W[0] u<< 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:

```
intXLEN_t __rv_kslraw_u(int32_t a, int32_t b);
```

# 41. 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	OP-P 1110111

Syntax:

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

**Purpose:** Perform 16-bit signed integer element saturating addition and 16-bit signed integer element saturating subtraction in a 32-bit chunk in parallel. 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 exceed 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[x] = SE17(Rs1.W[x].H[1]) + SE17(Rs2.W[x].H[1]);
res2[x] = SE17(Rs1.W[x].H[0]) - SE17(Rs2.W[x].H[0]);
for (res[x] in [res1[x], res2[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.W[x].H[1] = res1[x].H[0];
Rd.W[x].H[0] = res2[x].H[0];
for RV32, x=0
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

#### Intrinsic functions:

- Required:

```
uintXLEN_t __rv_kstas16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_kstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __rv_v_kstas16(int16x4_t a, int16x4_t b);
```

## 42. 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	OP-P 1110111

Syntax:

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

**Purpose:** Perform 16-bit signed integer element saturating subtraction and 16-bit signed integer element saturating addition in a 32-bit chunk in parallel. 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 exceed 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[x] = SE17(Rs1.W[x].H[1]) - SE17(Rs2.W[x].H[1]);
res2[x] = SE17(Rs1.W[x].H[0]) + SE17(Rs2.W[x].H[0]);
for (res[x] in [res1[x], res2[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.W[x].H[1] = res1[x].H[0];
Rd.W[x].H[0] = res2[x].H[0];
for RV32, x=0
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t __rv_kstsa16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_kstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t __rv_v_kstsa16(int16x4_t a, int16x4_t b);
```

## 43. KSUB64 (64-bit Signed Saturating Subtraction)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
KSUB64 1001001	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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).

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);
res65 = SE65(R[a_H].R[a_L]) - SE65(R[b_H].R[b_L]);
if (res65 > (2^63)-1) {
    res65 = (2^63)-1; OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63; OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

RV64:

```
res65 = SE65(Rs1) - SE65(Rs2);
if (res65 > (2^63)-1) {
    res65 = (2^63)-1; OV = 1;
} else if (res65 < -2^63) {
    res65 = -2^63; OV = 1;
}
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int64_t __rv_ksub64(int64_t a, int64_t b);
```

```
uintXLEN_t __rv_kabs8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

## 44. 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	OP-P 1110111

Syntax:

```
KSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the signed lower 16-bit content of two registers with Q15 saturation.

Description: The signed lower 16-bit content of Rs2 is subtracted from the signed lower 16-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:

```
a17 = SE17(Rs1.H[0]);  
b17 = SE17(Rs2.H[0]);  
t17 = a17 - b17;  
if (t17 > (2^15)-1) {  
    t17 = (2^15)-1;  
    OV = 1;  
} else if (t17 < -2^15) {  
    t17 = -2^15;  
    OV = 1  
}  
Rd = SE32(t17.H[0]); // RV32  
Rd = SE64(t17.H[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int32_t _rv_ksubh(int16_t a, int16_t b);
```

## 45. 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	OP-P 1110111

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]; // overflow ignored
```

RV64:

```
Mresult = Rs1.W[0] * Rs2.W[0];  
tres[31:0] = Rd.W[0] + Mresult.W[0]; // overflow ignored  
Rd = SE64(tres[31:0]);
```

Exceptions: None

Privilege level: All

**Note:** This instruction can be easily generated by a compiler without using the intrinsic function.

Intrinsic functions:

```
int32_t _rv_maddr32(int32_t t, int32_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

## 46. 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	OP-P 1110111

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]; // overflow ignored
```

RV64:

```
Mresult = Rs1.W[0] * Rs2.W[0];  
tres[31:0] = Rd.W[0] - Mresult.W[0]; // overflow ignored  
Rd = SE64(tres[31:0]);
```

Exceptions: None

Privilege level: All

Note: This instruction can be easily generated by a compiler without using the intrinsic function.

Intrinsic functions:

```
int32_t __rv_msabr32(int32_t t, int32_t a, int32_t b);
```

## 47. PKBB16, PKBT16, PKTT16, PKTB16

PKBB16 (Pack Two 16-bit Data from Both Bottom Half)

PKBT16 (Pack Two 16-bit Data from Bottom and Top Half)

PKTT16 (Pack Two 16-bit Data from Both Top Half)

Type: DSP

- PKBB16: RV32: Replaced with PACK in Zbpbo, RV64: Zpn
- PKTT16: RV32: Replaced with PACKU in Zbpbo, RV64: Zpn

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	OP-P 1110111

<u>xy</u>	<u>zz</u>	RV32	RV64
BB	00		✓
BT	01	✓	✓
TT	10		✓
TB	11	✓	✓

For RV32 PACK/PACKU encoding format, please see Section 6.7.

Syntax:

RV32:

PKBB16 Rd, Rs1, Rs2 == PACK Rd, Rs2, Rs1  
PKBT16 Rd, Rs1, Rs2  
PKTT16 Rd, Rs1, Rs2 == PACKU Rd, Rs2, Rs1  
PKTB16 Rd, Rs1, Rs2

RV64:

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].H[0] to Rd.W[x].H[1] and moves Rs2.W[x].H[0] to Rd.W[x].H[0].

(PKBT16) moves Rs1.W[x].H[0] to Rd.W[x].H[1] and moves Rs2.W[x].H[1] to Rd.W[x].H[0].

(PKTT16) moves Rs1.W[x].H[1] to Rd.W[x].H[1] and moves Rs2.W[x].H[1] to Rd.W[x].H[0].

(PKTB16) moves Rs1.W[x].H[1] to Rd.W[x].H[1] and moves Rs2.W[x].H[0] to Rd.W[x].H[0].

#### Operations:

```
Rd.W[x] = CONCAT(Rs1.W[x].H[0], Rs2.W[x].H[0]); // PKBB16
Rd.W[x] = CONCAT(Rs1.W[x].H[0], Rs2.W[x].H[1]); // PKBT16
Rd.W[x] = CONCAT(Rs1.W[x].H[1], Rs2.W[x].H[0]); // PKTB16
Rd.W[x] = CONCAT(Rs1.W[x].H[1], Rs2.W[x].H[1]); // PKTT16
for RV32: x=0,
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

#### Intrinsic functions:

- PKBB16
- Required:

```
uintXLEN_t _rv_pkbb16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

```
RV32:
uint16x2_t _rv_v_pkbb16(uint16x2_t a, uint16x2_t b);
RV64:
uint16x4_t _rv_v_pkbb16(uint16x4_t a, uint16x4_t b);
uintXLEN_t _rv_kabs8(uintXLEN_t a);
```

- PKBT16
- Required:

```
uintXLEN_t _rv_pkbt16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_pkbt16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_pkbt16(uint16x4_t a, uint16x4_t b);
```

- PKTB16

- Required:

```
uintXLEN_t __rv_pkbt16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_pkbt16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_pkbt16(uint16x4_t a, uint16x4_t b);
```

- PKTT16

- Required:

```
uintXLEN_t __rv_pktt16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_pktt16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_pktt16(uint16x4_t a, uint16x4_t b);
```

## 48. RDOV (Read OV flag)

Type: DSP

Format:

31 20	19 15	14 12	11 7	6 0
vxsat (0x009) 000000001001	00000	010	Rd	SYSTEM 1110011

Syntax:

```
RDOV Rd # pseudo mnemonic
```

Purpose: This pseudo instruction is an alias for "CSRR Rd, vxsat" instruction which maps to the real instruction of "CSRRS Rd, vxsat, x0".

Intrinsic functions:

```
uintXLEN_t __rv_rdov(void);
```

## 49. RSTAS16 (SIMD 16-bit Signed Averaging Straight Addition & Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
RSTAS16 1011010	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
RSTAS16 Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer element addition and 16-bit signed integer element subtraction in a 32-bit chunk in parallel. Operands are from corresponding positions in 32-bit chunks. The results are averaged 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:

```
res_add17[x] = (SE17(Rs1.W[x].H[1]) + SE17(Rs2.W[x].H[1])) s>> 1;  
res_sub17[x] = (SE17(Rs1.W[x].H[0]) - SE17(Rs2.W[x].H[0])) s>> 1;  
Rd.W[x].H[1] = res_add17[x].H[0];  
Rd.W[x].H[0] = res_sub17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.H" and "PASUB.H" instructions.

Intrinsic functions:

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t _rv_v_rstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t _rv_v_rstas16(int16x4_t a, int16x4_t b);
```

## 50. RSTSA16 (SIMD 16-bit Signed Averaging Straight Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
RSTSA16 1011011	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
RSTSA16 Rd, Rs1, Rs2
```

Purpose: Perform 16-bit signed integer element subtraction and 16-bit signed integer element addition in a 32-bit chunk in parallel. Operands are from corresponding positions in 32-bit chunks. The results are averaged 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:

```
res_sub17[x] = (SE17(Rs1.W[x].H[1]) - SE17(Rs2.W[x].H[1])) s>> 1;  
res_add17[x] = (SE17(Rs1.W[x].H[0]) + SE17(Rs2.W[x].H[0])) s>> 1;  
Rd.W[x].H[1] = res_sub17[x].H[0];  
Rd.W[x].H[0] = res_add17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.H" and "PASUB.H" instructions.

Intrinsic functions:

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t _rv_v_rstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
int16x4_t _rv_v_rstsa16(int16x4_t a, int16x4_t b);
```

# 51. RSUB64 (64-bit Signed Averaging Subtraction)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
RSUB64 1000001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
RSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit signed integer subtraction. The result is averaged to avoid overflow or saturation.

**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). The subtraction 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  $2d+1$ .

**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);
res65 = (SE65(R[a_H].R[a_L]) - SE65(R[b_H].R[b_L])) s>> 1;
R[t_H].R[t_L] = res65.D[0];
```

RV64:

```
Rd = (Rs1 - Rs2) s>> 1;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int64_t __rv_rsub64(int64_t a, int64_t b);
```

## 52. ABS (KABSW) (Scalar 32-bit Absolute Value)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ONEOP 1010110	ABS 10100	Rs1	000	Rd	OP-P 1110111

Syntax:

```
ABS Rd, Rs1
```

Purpose: Compute the absolute value of a signed 32-bit integer in a general purpose 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.

Operations:

```
if (Rs1.W[0] s>= 0) {  
    res = Rs1.W[0];  
} else {  
    res = -Rs1.W[0];  
}  
Rd = res;          // RV32  
Rd = SE64(res);   // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int32_t __rv_kabsw(int32_t a);
```

## 53. RADD64 (64-bit Signed Averaging Addition)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31 25	24 20	19 15	14 12	11 7	6 0
RADD64 1000000	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

RADD64 Rd, Rs1, Rs2

Purpose: Add two 64-bit signed integers. The result is averaged 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  $2d+1$ .

**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),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);
res65 = (SE65(R[a_H].R[a_L]) + SE65(R[b_H].R[b_L])) s>> 1;
R[t_H].R[t_L] = res65.D[0];
```

RV64:

```
res65 = (SE65(Rs1) + SE65(Rs2)) s>> 1;  
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
int64_t __rv_radd64(int64_t a, int64_t b);
```

## 54. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SCLIP8 Rd, Rs1, imm3u
```

Purpose: Limit the 8-bit signed integer elements of a register to a signed range in parallel.

Description: This instruction limits the 8-bit signed integer elements stored in Rs1 to a signed integer range between  $2^{\text{imm3u}}-1$  and  $-2^{\text{imm3u}}$ , 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:**

- Required:

```
uintXLEN_t __rv_sclip8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int8x4_t __rv_v_sclip8(int8x4_t a, uint32_t b);
```

RV64:

```
int8x8_t __rv_v_sclip8(int8x8_t a, uint32_t b);
```

## 55. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SCLIP16 Rd, Rs1, imm4u
```

Purpose: Limit the 16-bit signed integer elements of a register to a signed range in parallel.

Description: This instruction limits the 16-bit signed integer elements stored in Rs1 to a signed integer range between  $2^{\text{imm4u}}-1$  and  $-2^{\text{imm4u}}$ , 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:

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sclip16(int16x2_t a, uint32_t b);
```

RV64:

```
int16x4_t __rv_v_sclip16(int16x4_t a, uint32_t b);
```

## 56. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SCLIP32 Rd, Rs1, imm5u
```

Purpose: Limit the 32-bit signed integer elements of a register to a signed range in parallel.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 to a signed integer range between  $2^{\text{imm5u}}-1$  and  $-2^{\text{imm5u}}$ , 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:**

- Required:
- Optional (e.g., GCC vector extensions):

**RV64:**

```
int32x2_t __rv_v_sclip32(int32x2_t a, uint32_t b);
```

## 57. 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	OP-P 1110111

Syntax:

```
SLL8 Rd, Rs1, Rs2
```

Purpose: Perform 8-bit elements logical left shift operations in parallel. 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:

- Required:

```
uintXLEN_t _rv_sll8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t _rv_v_sll8(uint8x4_t a, uint32_t b);
```

RV64:

```
uint8x8_t _rv_v_sll8(uint8x8_t a, uint32_t b);
```

## 58. 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		Rs1		000		Rd		OP-P 1110111	

Syntax:

```
SLLI8 Rd, Rs1, imm3u
```

Purpose: Perform 8-bit elements logical left shift operations in parallel. 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;  
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:

- Required:

## 59. 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	OP-P 1110111

Syntax:

```
SLL16 Rd, Rs1, Rs2
```

Purpose: Perform 16-bit elements logical left shift operations in parallel. 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:

- Required:

```
uintXLEN_t _rv_sll16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
RV32:
```

```
  uint16x2_t _rv_v_sll16(uint16x2_t a, uint32_t b);
```

```
RV64:
```

```
  uint16x4_t _rv_v_sll16(uint16x4_t a, uint32_t b);
```

## 60. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SLLI16 Rd, Rs1, imm4u
```

Purpose: Perform 16-bit element logical left shift operations in parallel. 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 imm4u constant. And the results are written to Rd.

Operations:

```
sa = imm4u;  
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:

- Required:

# 61. SMAL (Signed Multiply Halfs & Add 64-bit)

Type: Partial-SIMD

Sub-extension: Zpsfoperand

Format:

31	25	24	20	19	15	14	12	11	7	6	0
SMAL 0101111		Rs2		Rs1		001		Rd		OP-P 1110111	

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.

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

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

**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.

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

**Operations:**

**RV32:**

```

Mres[31:0] = Rs2.H[1] s* 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);
// overflow ignored
R[Idx3].R[Idx2] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);

```

RV64:

```

Mres[0][31:0] = Rs2.W[0].H[1] s* Rs2.W[0].H[0];
Mres[1][31:0] = Rs2.W[1].H[1] s* Rs2.W[1].H[0];
// overflow ignored
Rd = Rs1 + SE64(Mres[1][31:0]) + SE64(Mres[0][31:0]);

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_smal(int64_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t __rv_v_smal(int64_t a, int16x2_t b);
```

RV64:

```
int64_t __rv_v_smal(int64_t a, int16x4_t b);
```

## 62. SMALBB, SMALBT, SMALTT

SMALBB (Signed Multiply Bottom Halfs & Add 64-bit)

SMALBT (Signed Multiply Bottom Half & Top Half & Add 64-bit)

SMALTT (Signed Multiply Top Halfs & Add 64-bit)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

### SMALBB

31 25	24 20	19 15	14 12	11 7	6 0
SMALBB 1000100	Rs2	Rs1	001	Rd	OP-P 1110111

### SMALBT

31 25	24 20	19 15	14 12	11 7	6 0
SMALBT 1001100	Rs2	Rs1	001	Rd	OP-P 1110111

### SMALTT

31 25	24 20	19 15	14 12	11 7	6 0
SMALTT 1010100	Rs2	Rs1	001	Rd	OP-P 1110111

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: rd pair + bottom\*bottom (all 32-bit elements)
- SMALBT rd pair + bottom\*top (all 32-bit elements)
- SMALTT rd 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.

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

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

## 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.

For this instruction, any potential overflow bit beyond 64-bit is discarded and ignored.

## Operations:

### RV32:

```
Mres[31:0] = Rs1.H[0] s* Rs2.H[0]; // SMALBB
Mres[31:0] = Rs1.H[0] s* Rs2.H[1]; // SMALBT
Mres[31:0] = Rs1.H[1] s* Rs2.H[1]; // SMALTT
Idx0 = CONCAT(Rd(4,1),1'b0); Idx1 = CONCAT(Rd(4,1),1'b1);
// overflow ignored
R[Idx1].R[Idx0] = R[Idx1].R[Idx0] + SE64(Mres[31:0]);
```

### RV64:

```
// SMALBB  
Mres[0][31:0] = Rs1.W[0].H[0] s* Rs2.W[0].H[0];  
Mres[1][31:0] = Rs1.W[1].H[0] s* Rs2.W[1].H[0];
```

```
// SMALBT  
Mres[0][31:0] = Rs1.W[0].H[0] s* Rs2.W[0].H[1];  
Mres[1][31:0] = Rs1.W[1].H[0] s* Rs2.W[1].H[1];
```

```
// SMALTT  
Mres[0][31:0] = Rs1.W[0].H[1] s* Rs2.W[0].H[1];  
Mres[1][31:0] = Rs1.W[1].H[1] s* Rs2.W[1].H[1];  
// overflow ignored  
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]);
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMALBB
- Required:

```
int64_t _rv_smalbb(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smalbb(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smalbb(int64_t t, int16x4_t a, int16x4_t b);
```

- SMALBT
- Required:

```
int64_t _rv_smalbt(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smalbt(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smalbt(int64_t t, int16x4_t a, int16x4_t b);
```

- SMALTT

- Required:

```
int64_t _rv_smaltt(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smaltt(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smaltt(int64_t t, int16x4_t a, int16x4_t b);
```

## 63. SMALDS, SMALDRS, SMALXDS

SMALDS (Signed Multiply Two Halfs & Subtract & Add 64-bit)

SMALDRS (Signed Multiply Two Halfs & Reverse Subtract & Add 64-bit)

SMALXDS (Signed Crossed Multiply Two Halfs & Subtract & Add 64-bit)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

SMALDS

31 25	24 20	19 15	14 12	11 7	6 0
SMALDS 1000101	Rs2	Rs1	001	Rd	OP-P 1110111

SMALDRS

31 25	24 20	19 15	14 12	11 7	6 0
SMALDRS 1001101	Rs2	Rs1	001	Rd	OP-P 1110111

SMALXDS

31 25	24 20	19 15	14 12	11 7	6 0
SMALXDS 1010101	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SMALDS Rd, Rs1, Rs2
SMALDRS Rd, Rs1, Rs2
SMALXDS Rd, Rs1, Rs2
```

Purpose: Perform 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. Next, 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: rd pair + (top\*top - bottom\*bottom) (all 32-bit elements)
- SMALDRS: rd pair + (bottom\*bottom - top\*top) (all 32-bit elements)
- SMALXDS: rd 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  $2d+1$ .

#### 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:

```
// Q31 = Q30 - Q30 = Q15*Q15 - Q15*Q15, overflow ignored
Mres[31:0] = (Rs1.H[1] s* Rs2.H[1]) - (Rs1.H[0] s* Rs2.H[0]); // SMALDS
Mres[31:0] = (Rs1.H[0] s* Rs2.H[0]) - (Rs1.H[1] s* Rs2.H[1]); // SMALDRS
Mres[31:0] = (Rs1.H[1] s* Rs2.H[0]) - (Rs1.H[0] s* 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]); // overflow ignored

```

- RV64:

```

// Q31 = Q30 - Q30 = Q15*Q15 - Q15*Q15, overflow ignored
// SMALDS
Mres[0][31:0] = (Rs1.W[0].H[1] s* Rs2.W[0].H[1]) - (Rs1.W[0].H[0] s*
Rs2.W[0].H[0]);
Mres[1][31:0] = (Rs1.W[1].H[1] s* Rs2.W[1].H[1]) - (Rs1.W[1].H[0] s*
Rs2.W[1].H[0]);

```

```

// SMALDRS
Mres[0][31:0] = (Rs1.W[0].H[0] s* Rs2.W[0].H[0]) - (Rs1.W[0].H[1] s*
Rs2.W[0].H[1]);
Mres[1][31:0] = (Rs1.W[1].H[0] s* Rs2.W[1].H[0]) - (Rs1.W[1].H[1] s*
Rs2.W[1].H[1]);

```

```

// SMALXDS
Mres[0][31:0] = (Rs1.W[0].H[1] s* Rs2.W[0].H[0]) - (Rs1.W[0].H[0] s*
Rs2.W[0].H[1]);
Mres[1][31:0] = (Rs1.W[1].H[1] s* Rs2.W[1].H[0]) - (Rs1.W[1].H[0] s*
Rs2.W[1].H[1]);

```

```
Rd = Rd + SE64(Mres[0][31:0]) + SE64(Mres[1][31:0]); // overflow ignored
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- SMALDS
- Required:

```
int64_t _rv_smalds(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smalds(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smalds(int64_t t, int16x4_t a, int16x4_t b);
```

- SMALDRS

- Required:

```
int64_t _rv_smaldrs(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smaldrs(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smaldrs(int64_t t, int16x4_t a, int16x4_t b);
```

- SMALXDS

- Required:

```
int64_t _rv_smalxds(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smalxds(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smalxds(int64_t t, int16x4_t a, int16x4_t b);
```

## 64. 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	OP-P 1110111

Syntax:

```
SMAQA Rd, Rs1, Rs2
```

Purpose: Perform 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.

For this instruction, any potential overflow bits beyond 32-bit are discarded and ignored.

Operations:

```
res[x] = Rd.W[x] +
    SE32(Rs1.W[x].B[3] s* Rs2.W[x].B[3]) + SE32(Rs1.W[x].B[2] s* Rs2.W[x].B[2]) +
    SE32(Rs1.W[x].B[1] s* Rs2.W[x].B[1]) + SE32(Rs1.W[x].B[0] s* Rs2.W[x].B[0]); // 
overflow discarded
Rd.W[x] = res[x];
for RV32: x=0,
for RV64: x=1,0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
intXLEN_t _rv_smaqa(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_smaqa(int32_t t, int8x4_t a, int8x4_t b);
```

RV64:

```
int32x2_t _rv_v_smaqa(int32x2_t t, int8x8_t a, int8x8_t b);
```

## 65. 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	OP-P 1110111

Syntax:

SMAQA.SU Rd, Rs1, Rs2

Purpose: Perform 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.

For this instruction, any potential overflow bits beyond 32-bit are discarded and ignored.

Operations:

```
res[x] = Rd.W[x] +
    SE32(Rs1.W[x].B[3] su* Rs2.W[x].B[3]) + SE32(Rs1.W[x].B[2] su* Rs2.W[x].B[2]) +
    SE32(Rs1.W[x].B[1] su* Rs2.W[x].B[1]) + SE32(Rs1.W[x].B[0] su* Rs2.W[x].B[0]);    //
overflow discarded
Rd.W[x] = res[x];
for RV32: x=0,
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
intXLEN_t __rv_smaqa_su(intXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t __rv_v_smaqa_su(int32_t t, int8x4_t a, uint8x4_t b);
```

RV64:

```
int32x2_t __rv_v_smaqa_su(int32x2_t t, int8x8_t a, uint8x8_t b);
```

## 66. SMDS, SMDRS, SMXDS

SMDS (SIMD Signed Multiply Two Halves and Subtract)

SMDRS (SIMD Signed Multiply Two Halves and Reverse Subtract)

SMXDS (SIMD Signed Crossed Multiply Two Halves and Subtract)

Type: SIMD

Format:

SMDS

31 25	24 20	19 15	14 12	11 7	6 0
SMDS 0101100	Rs2	Rs1	001	Rd	OP-P 1110111

SMDRS

31 25	24 20	19 15	14 12	11 7	6 0
SMDRS 0110100	Rs2	Rs1	001	Rd	OP-P 1110111

SMXDS

31 25	24 20	19 15	14 12	11 7	6 0
SMXDS 0111100	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

SMDS Rd, Rs1, Rs2

SMDRS Rd, Rs1, Rs2

SMXDS Rd, Rs1, Rs2

Purpose: Perform 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:

$$Rd.W[x] = (Rs1.W[x].H[1] s^* Rs2.W[x].H[1]) - (Rs1.W[x].H[0] s^* Rs2.W[x].H[0]);$$

- SMDRS:

$$Rd.W[x] = (Rs1.W[x].H[0] s^* Rs2.W[x].H[0]) - (Rs1.W[x].H[1] s^* Rs2.W[x].H[1]);$$

- SMXDS:

$$Rd.W[x] = (Rs1.W[x].H[1] s^* Rs2.W[x].H[0]) - (Rs1.W[x].H[0] s^* Rs2.W[x].H[1]);$$

Exceptions: None

Privilege level: All

#### Note:

- Usage domain: Complex, Statistics, Transform

#### Intrinsic functions:

- SMDS
- Required:

```
intXLEN_t _rv_smrs(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_smds(int16x2_t a, int16x2_t b);
```

RV64:

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

- SMDRS

- Required:

```
intXLEN_t _rv_smdrs(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_smdrs(int16x2_t a, int16x2_t b);
```

RV64:

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

- SMXDS

- Required:

```
intXLEN_t _rv_smxds(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int32_t _rv_v_smxds(int16x2_t a, int16x2_t b);
```

RV64:

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

## 67. SMSLDA, SMSLXDA

SMSLDA (Signed Multiply Two Halfs & Add & Subtract 64-bit)

SMSLXDA (Signed Crossed Multiply Two Halfs & Add & Subtract 64-bit)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

SMSLDA

31    25	24    20	19    15	14    12	11    7	6    0
SMSLDA 1010110	Rs2	Rs1	001	Rd	OP-P 1110111

SMSLXDA

31    25	24    20	19    15	14    12	11    7	6    0
SMSLXDA 1011110	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SMSLDA Rd, Rs1, Rs2
SMSLXDA Rd, Rs1, Rs2
```

**Purpose:** Perform 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  $2d+1$ .

## 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] s* Rs2.H[0]);
Mres1[31:0] = (Rs1.H[1] s* Rs2.H[1]);
// SMSLXDA
Mres0[31:0] = (Rs1.H[0] s* Rs2.H[1]);
Mres1[31:0] = (Rs1.H[1] s* 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] s* Rs2.W[0].H[0]);
Mres1[0][31:0] = (Rs1.W[0].H[1] s* Rs2.W[0].H[1]);
Mres0[1][31:0] = (Rs1.W[1].H[0] s* Rs2.W[1].H[0]);
Mres1[1][31:0] = (Rs1.W[1].H[1] s* Rs2.W[1].H[1]);
// SMSLXDA
Mres0[0][31:0] = (Rs1.W[0].H[0] s* Rs2.W[0].H[1]);
Mres1[0][31:0] = (Rs1.W[0].H[1] s* Rs2.W[0].H[0]);
Mres0[1][31:0] = (Rs1.W[1].H[0] s* Rs2.W[1].H[1]);
Mres1[1][31:0] = (Rs1.W[1].H[1] s* 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
- Required:

```
int64_t _rv_smlda(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smlda(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smlda(int64_t t, int16x4_t a, int16x4_t b);
```

- SMSLXDA
- Required:

```
int64_t _rv_smmlxda(int64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int64_t _rv_v_smmlxda(int64_t t, int16x2_t a, int16x2_t b);
```

RV64:

```
int64_t _rv_v_smmlxda(int64_t t, int16x4_t a, int16x4_t b);
```

## 68. SMSR64 (Signed Multiply and Subtract from 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
SMSR64 1000011	Rs2	Rs1	001	Rd	OP-P 1110111

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  $2d+1$ .

**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:

$d_L = \text{CONCAT}(\text{Rd}(4,1), 1'b0); d_H = \text{CONCAT}(\text{Rd}(4,1), 1'b1);$   
 $R[d_H].R[d_L] = R[d_H].R[d_L] - (\text{Rs1 } s^* \text{ Rs2});$

- RV64:

$Rd = Rd - (\text{Rs1.W[0] } s^* \text{ Rs2.W[0]} - (\text{Rs1.W[1] } s^* \text{ Rs2.W[1]});$

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_smsr64(int64_t t, intXLEN_t a, intXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
int64_t __rv_v_smsr64(int64_t t, int32x2_t a, int32x2_t b);
```

## 69. SMUL8, SMULX8

### SMUL8 (SIMD Signed 8-bit Multiply)

#### SMULX8 (SIMD Signed Crossed 8-bit Multiply)

Type: SIMD

Sub-extension: Zpsfoperand

Format:

SMUL8

31 25	24 20	19 15	14 12	11 7	6 0
SMUL8 1010100	Rs2	Rs1	000	Rd	OP-P 1110111

SMULX8

31 25	24 20	19 15	14 12	11 7	6 0
SMULX8 1010101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

**SMUL8 Rd, Rs1, Rs2**  
**SMULX8 Rd, Rs1, Rs2**

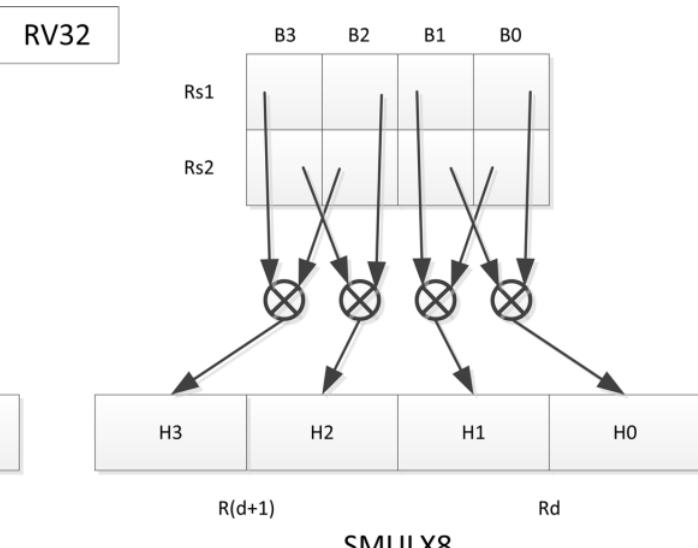
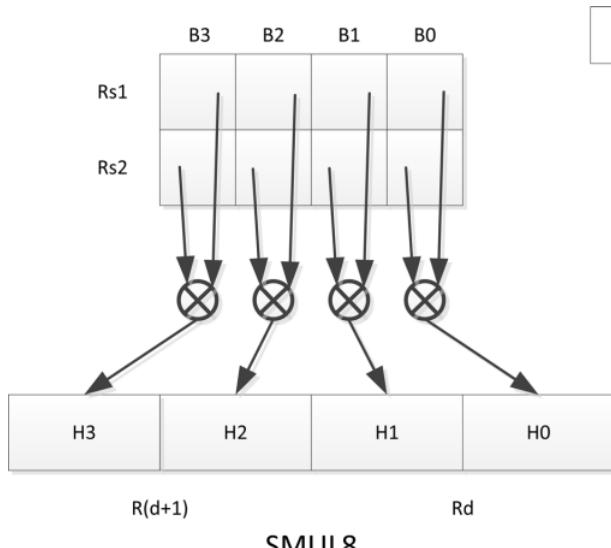
Purpose: Perform signed 8-bit multiplications and generate four 16-bit results in parallel.

**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*.

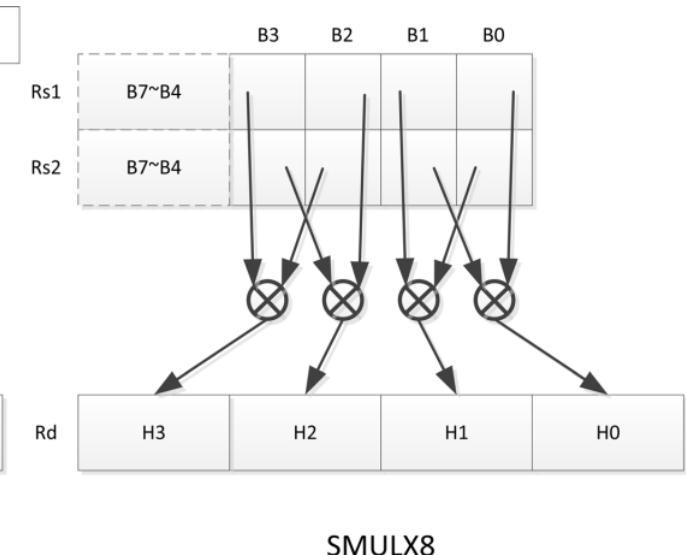
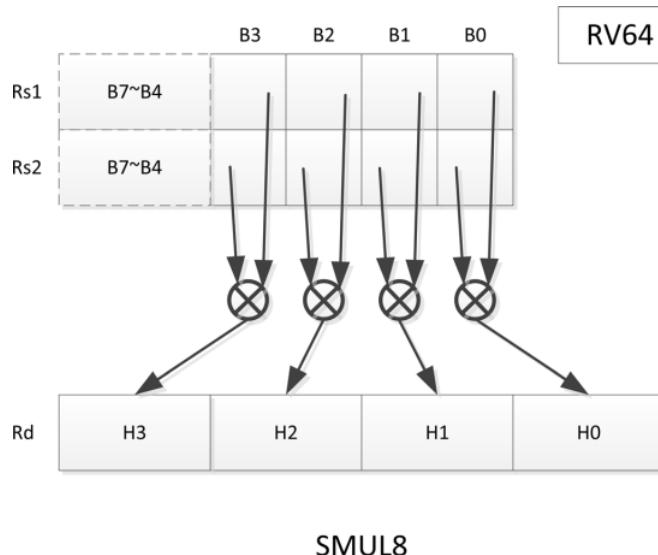
The odd "*2d+1*" register of the pair contains the two 16-bit results calculated from the top part of Rs1 and the even "*2d*" register of the pair contains the two 16-bit results calculated from the bottom part of Rs1.



**RV64 Description:** For the “SMUL8” instruction, multiply the 8-bit data elements of the lower 32-bit word of Rs1 with the corresponding 8-bit data elements of the lower 32-bit word of Rs2.

For the “SMULX8” instruction, multiply the *first* and *second* 8-bit data elements of the lower 32-bit word of Rs1 with the *second* and *first* 8-bit data elements of the lower 32-bit word of Rs2. At the same time, multiply the *third* and *fourth* 8-bit data elements of the lower 32-bit word of Rs1 with the *fourth* and *third* 8-bit data elements of the lower 32-bit word 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:

```

d_L = CONCAT(Rd(4,1),1'b0); d_H = CONCAT(Rd(4,1),1'b1);
// SMUL8
R[d_L].H[0] = Rs1.B[0] s* Rs2.B[0];
R[d_L].H[1] = Rs1.B[1] s* Rs2.B[1];
R[d_H].H[0] = Rs1.B[2] s* Rs2.B[2];
R[d_H].H[1] = Rs1.B[3] s* Rs2.B[3];
// SMULX8
R[d_L].H[0] = Rs1.B[0] s* Rs2.B[1];
R[d_L].H[1] = Rs1.B[1] s* Rs2.B[0];
R[d_H].H[0] = Rs1.B[2] s* Rs2.B[3];
R[d_H].H[1] = Rs1.B[3] s* Rs2.B[2];

```

- RV64:

```

// SMUL8
Rd.W[0].H[0] = Rs1.B[0] s* Rs2.B[0];
Rd.W[0].H[1] = Rs1.B[1] s* Rs2.B[1];
Rd.W[1].H[0] = Rs1.B[2] s* Rs2.B[2];
Rd.W[1].H[1] = Rs1.B[3] s* Rs2.B[3];
// SMULX8
Rd.W[0].H[0] = Rs1.B[0] s* Rs2.B[1];
Rd.W[0].H[1] = Rs1.B[1] s* Rs2.B[0];
Rd.W[1].H[0] = Rs1.B[2] s* Rs2.B[3];
Rd.W[1].H[1] = Rs1.B[3] s* Rs2.B[2];

```

**Exceptions:** None

**Privilege level:** All

**Note:**

**Intrinsic functions:**

- SMUL8
- Required:

```
uint64_t __rv_smul8(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int16x4_t __rv_v_smul8(int8x4_t a, int8x4_t b);
```

- SMULX8
- Required:

```
uint64_t __rv_smulx8(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int16x4_t _rv_v_smulx8(int8x4_t a, int8x4_t b);
```

## 70. SMUL16, SMULX16

### SMUL16 (SIMD Signed 16-bit Multiply)

#### SMULX16 (SIMD Signed Crossed 16-bit Multiply)

Type: SIMD

Sub-extension: Zpsfoperand

Format:

**SMUL16**

31    25	24    20	19    15	14    12	11    7	6    0
SMUL16 1010000	Rs2	Rs1	000	Rd	OP-P 1110111

**SMULX16**

31    25	24    20	19    15	14    12	11    7	6    0
SMULX16 1010001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

**SMUL16 Rd, Rs1, Rs2**  
**SMULX16 Rd, Rs1, Rs2**

Purpose: Perform signed 16-bit multiplications and generate two 32-bit results in parallel.

**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:

```
d_L = CONCAT(Rd(4,1),1'b0); d_H = CONCAT(Rd(4,1),1'b1);
// SMUL16
R[d_H] = Rs1.H[1] s* Rs2.H[1];
R[d_L] = Rs1.H[0] s* Rs2.H[0];
// SMULX16
R[d_H] = Rs1.H[1] s* Rs2.H[0];
R[d_L] = Rs1.H[0] s* Rs2.H[1];
```

- RV64:

```
// SMUL16
Rd.W[1] = Rs1.H[1] s* Rs2.H[1];
Rd.W[0] = Rs1.H[0] s* Rs2.H[0];
// SMULX16
Rd.W[1] = Rs1.H[1] s* Rs2.H[0];
Rd.W[0] = Rs1.H[0] s* Rs2.H[1];
```

Exceptions: None

Privilege level: All

Note:

#### Intrinsic functions:

- SMUL16
- Required:

```
int64_t __rv_smul16(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_smul16(int16x2_t a, int16x2_t b);
```

- SMULX16

- Required:

```
int64_t _rv_smulx16(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t _rv_v_smulx16(int16x2_t a, int16x2_t b);
```

# 71. 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	OP-P 1110111

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:

```
intXLEN_t __rv_sra_u(intXLEN_t a, uint32_t b);
```

## 72. SRAI.u (Rounding Shift Right Arithmetic Immediate)

Type: DSP

Format:

RV32

31 25	24 20	19 15	14 12	11 7	6 0
SRAI.u 1101010	imm5u	Rs1	001	Rd	OP-P 1110111

RV64

31 26	25 20	19 15	14 12	11 7	6 0
SRAI.u 110101	imm6u	Rs1	001	Rd	OP-P 1110111

Syntax:

```
SRAI.u Rd, Rs1, imm5u (RV32)
SRAI.u Rd, Rs1, imm6u (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 imm5u (RV32) or imm6u (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 = imm5u;
if (sa != 0) {
    res[31:-1] = SE33(Rs1[31:(sa-1)]) + 1;
    Rd = res[31:0];
} else {
    Rd = Rs1;
}
```

- RV64:

```
sa = imm6u;
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:

```
intXLEN_t __rv_sra_u(intXLEN_t a, uint32_t b);
```

## 73. SRA8, SRA8.u

### SRA8 (SIMD 8-bit Shift Right Arithmetic)

#### 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	OP-P 1110111

SRA8.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA8.u 0110100	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

SRA8 Rd, Rs1, Rs2  
SRA8.u Rd, Rs1, Rs2

**Purpose:** Perform 8-bit element arithmetic right shift operations in parallel. 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
- Required:

```
uintXLEN_t __rv_sra8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int8x4_t __rv_v_sra8(int8x4_t a, uint32_t b);
RV64:
int8x8_t __rv_v_sra8(int8x8_t a, uint32_t b);

```

- SRA8.u
- Required:

```
uintXLEN_t __rv_sra8_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int8x4_t __rv_v_sra8_u(int8x4_t a, uint32_t b);
RV64:
int8x8_t __rv_v_sra8_u(int8x8_t a, uint32_t b);

```

## 74. SRAI8, SRAI8.u

SRAI8 (SIMD 8-bit Shift Right Arithmetic Immediate)

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	Rs1	000	Rd	OP-P 1110111

SRAI8.u

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRAI8.u 0111100	01	imm3u	Rs1	000	Rd	OP-P 1110111

Syntax:

SRAI8 Rd, Rs1, imm3u  
SRAI8.u Rd, Rs1, imm3u

**Purpose:** Perform 8-bit element arithmetic right shift operations in parallel. 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;
if (sa != 0) {
    if ('.u' form) { // SRAI8.u
        res[7:-1] = SE9(Rs1.B[x][7:sa-1]) + 1;
        Rd.B[x] = res[7:0];
    } else { // SRAI8
        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
- Required:

```
uintXLEN_t __rv_sra8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int8x4_t __rv_v_sra8(int8x4_t a, uint32_t b);
RV64:
int8x8_t __rv_v_sra8(int8x8_t a, uint32_t b);

```

- SRAI8.u
- Required:

```
uintXLEN_t __rv_sra8_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int8x4_t __rv_v_sra8_u(int8x4_t a, uint32_t b);
RV64:
int8x8_t __rv_v_sra8_u(int8x8_t a, uint32_t b);

```

## 75. SRA16, SRA16.u

### SRA16 (SIMD 16-bit Shift Right Arithmetic)

#### 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	OP-P 1110111

SRA16.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA16.u 0110000	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SRA16 Rd, Rs1, Rs2
SRA16.u Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit element arithmetic right shift operations in parallel. 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
- Required:

```
uintXLEN_t __rv_sra16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int16x2_t __rv_v_sra16(int16x2_t a, uint32_t b);
RV64:
int16x4_t __rv_v_sra16(int16x4_t a, uint32_t b);

```

- SRA16.u
- Required:

```
uintXLEN_t __rv_sra16_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int16x2_t __rv_v_sra16_u(int16x2_t a, uint32_t b);
RV64:
int16x4_t __rv_v_sra16_u(int16x4_t a, uint32_t b);

```

## 76. SRAI16, SRAI16.u

### SRAI16 (SIMD 16-bit Shift Right Arithmetic Immediate)

**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	imm4u	Rs1	000	Rd	OP-P 1110111

**SRAI16.u**

31 25	24	23 20	19 15	14 12	11 7	6 0
SRAI16.u 0111000	1	imm4u	Rs1	000	Rd	OP-P 1110111

Syntax:

**SRAI16 Rd, Rs1, imm4u**  
**SRAI16.u Rd, Rs1, imm4u**

**Purpose:** Perform 16-bit elements arithmetic right shift operations in parallel. 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;
if (sa != 0) {
    if (“.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
- Required:

```
uintXLEN_t __rv_sra16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int16x2_t __rv_v_sra16(int16x2_t a, uint32_t b);
RV64:
int16x4_t __rv_v_sra16(int16x4_t a, uint32_t b);

```

- SRAI16.u
- Required:

```
uintXLEN_t __rv_sra16_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```

RV32:
int16x2_t __rv_v_sra16_u(int16x2_t a, uint32_t b);
RV64:
int16x4_t __rv_v_sra16_u(int16x4_t a, uint32_t b);

```

## 77. SRL8, SRL8.u

### SRL8 (SIMD 8-bit Shift Right Logical)

#### 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	OP-P 1110111

SRL8.u

31    25	24    20	19    15	14    12	11    7	6    0
SRL8.u 0110101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SRL8 Rd, Rs1, Rs2
SRL8.u Rd, Rs1, Rs2
```

**Purpose:** Perform 8-bit elements logical right shift operations in parallel. 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 (“.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
- Required:

```
uintXLEN_t _rv_srl8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t _rv_v_srl8(uint8x4_t a, uint32_t b);
```

RV64:

```
uint8x8_t _rv_v_srl8(uint8x8_t a, uint32_t b);
```

- SRL8.u
- Required:

```
uintXLEN_t _rv_srl8_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t _rv_v_srl8_u(uint8x4_t a, uint32_t b);
```

RV64:

```
uint8x8_t _rv_v_srl8_u(uint8x8_t a, uint32_t b);
```

## 78. SRLI8, SRLI8.u

SRLI8 (SIMD 8-bit Shift Right Logical Immediate)

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	Rs1	000	Rd	OP-P 1110111

SRLI8.u

31 25	24 23	22 20	19 15	14 12	11 7	6 0
SRLI8.u 0111101	01	imm3u	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SRLI8 Rd, Rs1, imm3u
SRLI8.u Rd, Rs1, imm3u
```

**Purpose:** Perform 8-bit elements logical right shift operations in parallel. 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;
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

- Required:

```
uintXLEN_t __rv_srl8(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t __rv_v_srl8(uint8x4_t a, uint32_t b);
```

RV64:

```
uint8x8_t __rv_v_srl8(uint8x8_t a, uint32_t b);
```

- SRLI8.u

- Required:

```
uintXLEN_t __rv_srl8_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint8x4_t __rv_v_srl8_u(uint8x4_t a, uint32_t b);
```

RV64:

```
uint8x8_t __rv_v_srl8_u(uint8x8_t a, uint32_t b);
```

## 79. SRL16, SRL16.u

### SRL16 (SIMD 16-bit Shift Right Logical)

#### 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	OP-P 1110111

SRL16.u

31 25	24 20	19 15	14 12	11 7	6 0
SRL16.u 0110001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SRL16 Rd, Rs1, Rs2
SRL16.u Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit elements logical right shift operations in parallel. 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
- Required:

```
uintXLEN_t __rv_srl16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_srl16(uint16x2_t a, uint32_t b);
```

RV64:

```
uint16x4_t __rv_v_srl16(uint16x4_t a, uint32_t b);
```

- SRL16.u
- Required:

```
uintXLEN_t __rv_srl16_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_srl16_u(uint16x2_t a, uint32_t b);
```

RV64:

```
uint16x4_t __rv_v_srl16_u(uint16x4_t a, uint32_t b);
```

## 80. SRLI16, SRLI16.u

SRLI16 (SIMD 16-bit Shift Right Logical Immediate)

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	imm4u	Rs1	000	Rd	OP-P 1110111

SRLI16.u

31 25	24	23 20	19 15	14 12	11 7	6 0
SRLI16.u 0111001	1	imm4u	Rs1	000	Rd	OP-P 1110111

Syntax:

```
SRLI16 Rd, Rs1, imm4u
SRLI16.u Rd, Rs1, imm4u
```

**Purpose:** Perform 16-bit elements logical right shift operations in parallel. 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

- Required:

```
uintXLEN_t __rv_srl16(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_srl16(uint16x2_t a, uint32_t b);
```

RV64:

```
uint16x4_t __rv_v_srl16(uint16x4_t a, uint32_t b);
```

- SRLI16.u

- Required:

```
uintXLEN_t __rv_srl16_u(uintXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_srl16_u(uint16x2_t a, uint32_t b);
```

RV64:

```
uint16x4_t __rv_v_srl16_u(uint16x4_t a, uint32_t b);
```

# 81. 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	OP-P 1110111

Syntax:

```
STAS16 Rd, Rs1, Rs2
```

Purpose: Perform 16-bit integer element addition and 16-bit integer element subtraction in a 32-bit chunk in parallel. 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.

For this instruction, any potential overflow bit(s) during the computation are discarded and ignored.

Operations:

```
Rd.W[x].H[1] = Rs1.W[x].H[1] + Rs2.W[x].H[1]; // overflow discarded
Rd.W[x].H[0] = Rs1.W[x].H[0] - Rs2.W[x].H[0]; // overflow discarded
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:

- Required:

```
uintXLEN_t __rv_stas16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_ustas16(uint16x2_t a, uint16x2_t b);  
int16x2_t _rv_v_sstas16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_ustas16(uint16x4_t a, uint16x4_t b);  
int16x4_t _rv_v_sstas16(int16x4_t a, int16x4_t b);
```

## 82. 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	OP-P 1110111

Syntax:

```
STSA16 Rd, Rs1, Rs2
```

Purpose: Perform 16-bit integer element subtraction and 16-bit integer element addition in a 32-bit chunk in parallel. 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].H[1] = Rs1.W[x].H[1] - Rs2.W[x].H[1];  
Rd.W[x].H[0] = Rs1.W[x].H[0] + Rs2.W[x].H[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:

- Required:

```
uintXLEN_t __rv_stsa16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_ustsa16(uint16x2_t a, uint16x2_t b);  
int16x2_t __rv_v_sstsa16(int16x2_t a, int16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_ustsa16(uint16x4_t a, uint16x4_t b);  
int16x4_t __rv_v_sstsa16(int16x4_t a, int16x4_t b);
```

### 83. SUNPKD810, SUNPKD820, SUNPKD830, SUNPKD831, SUNPKD832

SUNPKD810 (Signed Unpacking Bytes 1 & 0)

SUNPKD820 (Signed Unpacking Bytes 2 & 0)

SUNPKD830 (Signed Unpacking Bytes 3 & 0)

SUNPKD831 (Signed Unpacking Bytes 3 & 1)

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	OP-P 1110111

<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
- Required:

```
uintXLEN_t __rv_sunpkd810(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sunpkd810(int8x4_t a);
```

RV64:

```
int16x4_t __rv_v_sunpkd810(int8x8_t a);
```

- SUNPK820
- Required:

```
uintXLEN_t __rv_sunpkd820(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sunpkd820(int8x4_t a);
```

RV64:

```
int16x4_t __rv_v_sunpkd820(int8x8_t a);
```

- SUNPK830
- Required:

```
uintXLEN_t __rv_sunpkd830(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sunpkd830(int8x4_t a);
```

RV64:

```
int16x4_t __rv_v_sunpkd830(int8x8_t a);
```

- SUNPK831

- Required:

```
uintXLEN_t __rv_sunpkd831(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sunpkd831(int8x4_t a);
```

RV64:

```
int16x4_t __rv_v_sunpkd831(int8x8_t a);
```

- SUNPK832

- Required:

```
uintXLEN_t __rv_sunpkd832(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
int16x2_t __rv_v_sunpkd832(int8x4_t a);
```

RV64:

```
int16x4_t __rv_v_sunpkd832(int8x8_t a);
```

## 84. SWAP8 (Swap Byte within Halfword)

Type: DSP

Replaced with REV8.H in Zbpbo extension.

Format:

31    25	24    20	19    15	14    12	11    7	6    0
ONEOP 1010110	SWAP8 11000	Rs1	000	Rd	OP-P 1110111

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].B[0],Rs1.H[x].B[1]);  
for RV32: x=1..0,  
for RV64: x=3..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t _rv_swap8(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

```
RV32:  
  uint8x4_t _rv_v_swap8(uint8x4_t a);  
RV64:  
  uint8x8_t _rv_v_swap8(uint8x8_t a);
```

## 85. SWAP16 (Swap Halfword within Word)

Type: DSP

Format:

31    25	24    20	19    15	14    12	11    7	6    0
PKBT16 0001111	Rs1	Rs1	001	Rd	OP-P 1110111

- An alias for "PKBT16 Rd, Rs1, Rs1"

Syntax:

```
SWAP16 Rd, Rs1 # pseudo mnemonic
```

**Purpose:** Swap the 16-bit halfwords within each word of a register. This pseudo instruction is an alias for "PKBT16 Rd, Rs1, Rs1" instruction.

**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].H[0],Rs1.W[x].H[1]);  
for RV32: x=0,  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t __rv_swap16(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

```
RV32:  
    uint16x2_t __rv_v_swap16(uint16x2_t a);  
RV64:  
    uint16x4_t __rv_v_swap16(uint16x4_t a);
```

## 86. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
UCLIP8 Rd, Rs1, imm3u
```

Purpose: Limit the 8-bit signed elements of a register to an unsigned range in parallel.

Description: This instruction limits the 8-bit signed elements stored in Rs1 to 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:

- Required:

```
RV32:
    uint8x4_t _rv_v_uclip8(int8x4_t a, uint32_t b);
RV64:
    uint8x8_t _rv_v_uclip8(int8x8_t a, uint32_t b);
```

## 87. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
UCLIP16 Rd, Rs1, imm4u
```

Purpose: Limit the 16-bit signed elements of a register to an unsigned range in parallel.

Description: This instruction limits the 16-bit signed elements stored in Rs1 to 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:

- Required:

```
RV32:
```

```
uint16x2_t _rv_v_uclip16(int16x2_t a, uint32_t b);
```

```
RV64:
```

```
uint16x4_t _rv_v_uclip16(int16x4_t a, uint32_t b);
```

## 88. 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	Rs1	000	Rd	OP-P 1110111

Syntax:

```
UCLIP32 Rd, Rs1, imm5u
```

Purpose: Limit the 32-bit signed integer elements of a register to an unsigned range in parallel.

Description: This instruction limits the 32-bit signed integer elements stored in Rs1 to 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:

- Required:

```
uintXLEN_t _rv_uclip32(intXLEN_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint32x2_t _rv_v_uclip32(int32x2_t a, uint32_t b);
```

## 89. UKADD64 (64-bit Unsigned Saturating Addition)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UKADD64 1011000	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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:

```
d_L = CONCAT(Rd(4,1),1'b0); d_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);
res65 = ZE65(R[a_H].R[a_L]) + ZE65(R[b_H].R[b_L]);
if (res65 > (2^64)-1) {
    res65 = (2^64)-1; OV = 1;
}
R[d_H].R[d_L] = res65.D[0];
```

- RV64:

```
res65 = ZE65(Rs1) + ZE65(Rs2);
if (res65 > (2^64)-1) {
    res65 = (2^64)-1; OV = 1;
}
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint64_t __rv_ukadd64(uint64_t a, uint64_t b);
```

## 90. 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	OP-P 1110111

Syntax:

```
UKADDH Rd, Rs1, Rs2
```

Purpose: Add the unsigned lower 16-bit content of two registers with U16 saturation.

Description: The unsigned lower 16-bit content of Rs1 is added with the unsigned lower 16-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:

```
a17 = ZE17(Rs1.H[0]);  
b17 = ZE17(Rs2.H[0]);  
t17 = a17 + b17;  
if (t17 u> (2^16)-1) {  
    t17 = (2^16)-1;  
    OV = 1;  
}  
Rd = SE32(t17.H[0]); // RV32  
Rd = SE64(t17.H[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint32_t __rv_ukaddh(uint16_t a, uint16_t b);
```

# 91. 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	OP-P 1110111

Syntax:

```
UKCRAS16 Rd, Rs1, Rs2
```

**Purpose:** Perform one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk in parallel. 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 exceed 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[x] = ZE17(Rs1.W[x].H[1]) + ZE17(Rs2.W[x].H[0]);  
res2[x] = ZE17(Rs1.W[x].H[0]) - ZE17(Rs2.W[x].H[1]);  
if (res1[x] u> (2^16)-1) {  
    res1[x] = (2^16)-1;  
    OV = 1;  
}  
if (res2[x] s< 0) {  
    res2[x] = 0;  
    OV = 1;  
}  
Rd.W[x].H[1] = res1[x].H[0];  
Rd.W[x].H[0] = res2[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t _rv_ukcras16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_ukcras16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_ukcras16(uint16x4_t a, uint16x4_t b);
```

## 92. UKCRSA16 (SIMD 16-bit Unsigned Saturating Cross Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UKCRSA16 0011011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
UKCRSA16 Rd, Rs1, Rs2
```

**Purpose:** Perform one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk in parallel. 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 exceed 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[x] = ZE17(Rs1.W[x].H[1]) - ZE17(Rs2.W[x].H[0]);
res2[x] = ZE17(Rs1.W[x].H[0]) + ZE17(Rs2.W[x].H[1]);
if (res1[x] < 0) {
    res1[x] = 0;
    OV = 1;
} else if (res2[x] > (2^16)-1) {
    res2[x] = (2^16)-1;
    OV = 1;
}
Rd.W[x].H[1] = res1[x].H[0];
Rd.W[x].H[0] = res2[x].H[0];
for RV32, x=0
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- Required:

```
uintXLEN_t _rv_ukcrsa16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_ukcrsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_ukcrsa16(uint16x4_t a, uint16x4_t b);
```

## 93. UKMAR64 (Unsigned Multiply and Saturating Add to 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UKMAR64 1011010	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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);
mul65 = ZE65(Rs1 u* Rs2);
top65 = ZE65(R[t_H].R[t_L]);
res65 = top65 + mul65;
if (res65 u> (2^64)-1) {
    res65 = (2^64)-1;
    OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

- RV64:

```

mula66 = ZE66(Rs1.W[0] u* Rs2.W[0]);
mulb66 = ZE66(Rs1.W[1] u* Rs2.W[1]);
res66 = ZE66(Rd) + mula66 + mulb66;
if (res66 u> (2^64)-1) {
    res66 = (2^64)-1;
    OV = 1;
}
Rd = res66.D[0];

```

**Exceptions:** None

**Privilege level:** All

**Note:**

**Intrinsic functions:**

- Required:

```
uint64_t __rv_ukmar64(uint64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

**RV64:**

```
uint64_t __rv_v_ukmar64(uint64_t t, uint32x2_t a, uint32x2_t b);
```

## 94. UKMSR64 (Unsigned Multiply and Saturating Subtract from 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UKMSR64 1011011	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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);
mul65 = ZE65(Rs1 u* Rs2);
top65 = ZE65(R[t_H].R[t_L]);
res65 = top65 - mul65;
if (res65 < 0) {
    res65 = 0;
    OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

- RV64:

```
mula66 = ZE66(Rs1.W[0] u* Rs2.W[0]);  
mulb66 = ZE66(Rs1.W[1] u* Rs2.W[1]);  
res66 = ZE66(Rd) - mula66 - mulb66;  
if (res66 < 0) {  
    res66 = 0;  
    OV = 1;  
}  
Rd = res66.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_ukmsr64(uint64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint64_t __rv_v_ukmsr64(uint64_t t, uint32x2_t a, uint32x2_t b);
```

## 95. 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	OP-P 1110111

Syntax:

```
UKSTAS16 Rd, Rs1, Rs2
```

**Purpose:** Perform one 16-bit unsigned integer element saturating addition and one 16-bit unsigned integer element saturating subtraction in a 32-bit chunk in parallel. 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 exceed 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[x] = ZE17(Rs1.W[x].H[1]) + ZE17(Rs2.W[x].H[1]);  
res2[x] = ZE17(Rs1.W[x].H[0]) - ZE17(Rs2.W[x].H[0]);  
if (res1[x] u> (2^16)-1) {  
    res1[x] = (2^16)-1;  
    OV = 1;  
}  
if (res2[x] s< 0) {  
    res2[x] = 0;  
    OV = 1;  
}  
Rd.W[x].H[1] = res1[x].H[0];  
Rd.W[x].H[0] = res2[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uintXLEN_t _rv_ukstas16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_ukstas16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_ukstas16(uint16x4_t a, uint16x4_t b);
```

## 96. 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	OP-P 1110111

Syntax:

```
UKSTSA16 Rd, Rs1, Rs2
```

**Purpose:** Perform one 16-bit unsigned integer element saturating subtraction and one 16-bit unsigned integer element saturating addition in a 32-bit chunk in parallel. 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 exceed 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[x] = ZE17(Rs1.W[x].H[1]) - ZE17(Rs2.W[x].H[1]);
res2[x] = ZE17(Rs1.W[x].H[0]) + ZE17(Rs2.W[x].H[0]);
if (res1[x] < 0) {
    res1[x] = 0;
    OV = 1;
} else if (res2[x] > (2^16)-1) {
    res2[x] = (2^16)-1;
    OV = 1;
}
Rd.W[x].H[1] = res1[x].H[0];
Rd.W[x].H[0] = res2[x].H[0];
for RV32, x=0
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

## Intrinsic functions:

- Required:

```
uintXLEN_t __rv_ukstsa16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_ukstsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_ukstsa16(uint16x4_t a, uint16x4_t b);
```

## 97. UKSUB64 (64-bit Unsigned Saturating Subtraction)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UKSUB64 1011001	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceeds 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  $2d+1$ .

**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 exceeds 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);
res65 = ZE65(R[a_H].R[a_L]) - ZE65(R[b_H].R[b_L]);
if (res65 < 0) {
    res65 = 0; OV = 1;
}
R[t_H].R[t_L] = res65.D[0];
```

- RV64

```
result = Rs1 - Rs2;
if (result < 0) {
    result = 0; OV = 1;
}
Rd = result;
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint64_t __rv_uksub64(uint64_t a, uint64_t b);
```

## 98. 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	OP-P 1110111

Syntax:

```
UKSUBH Rd, Rs1, Rs2
```

Purpose: Subtract the unsigned lower 16-bit content of two registers with U16 saturation.

Description: The unsigned lower 16-bit content of Rs2 is subtracted from the unsigned lower 16-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:

```
a17 = ZE17(Rs1.H[0]);  
b17 = ZE17(Rs2.H[0]);  
t17 = a17 - b17;  
if (t17 > (2^16)-1) {  
    t17 = (2^16)-1;  
    OV = 1;  
}  
else if (t17 < 0) {  
    t17 = 0;  
    OV = 1;  
}  
Rd = SE32(t17.H[0]); // RV32  
Rd = SE64(t17.H[0]); // RV64
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint32_t __rv_uksubh(uint16_t a, uint16_t b);
```

## 99. UMAR64 (Unsigned Multiply and Add to 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31      25	24      20	19      15	14      12	11      7	6      0
UMAR64 1010010	Rs2	Rs1	001	Rd	OP-P 1110111

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  $2d+1$ .

**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:

$$d_L = \text{CONCAT}(Rd(4,1), 1'b0); d_H = \text{CONCAT}(Rd(4,1), 1'b1);$$
$$R[d_H].R[d_L] = R[d_H].R[d_L] + (Rs1 \ u^* \ Rs2);$$

- 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:

- Required:

```
uint64_t __rv_umar64(uint64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint64_t __rv_v_umar64(uint64_t t, uint32x2_t a, uint32x2_t b);
```

# 100. 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	OP-P 1110111

Syntax:

UMAQA Rd, Rs1, Rs2

Purpose: Perform 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:

- Required:

```
uintXLEN_t _rv_umaqa(uintXLEN_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint32_t __rv_v_umaqa(uint32_t t, uint8x4_t a, uint8x4_t b);
```

RV64:

```
uint32x2_t __rv_v_umaqa(uint32x2_t t, uint8x8_t a, uint8x8_t b);
```

# 101. UMSR64 (Unsigned Multiply and Subtract from 64-Bit Data)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
UMSR64 1010011	Rs2	Rs1	001	Rd	OP-P 1110111

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  $2d+1$ .

**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:

$$d_L = \text{CONCAT}(Rd(4,1), 1'b0); d_H = \text{CONCAT}(Rd(4,1), 1'b1);$$
$$R[d_H].R[d_L] = R[d_H].R[d_L] - (Rs1 \ u^* \ Rs2);$$

- 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:

- Required:

```
uint64_t __rv_umsr64(uint64_t t, uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV64:

```
uint64_t __rv_v_umsr64(uint64_t t, uint32x2_t a, uint32x2_t b);
```

## 102. UMUL8, UMULX8

### UMUL8 (SIMD Unsigned 8-bit Multiply)

#### UMULX8 (SIMD Unsigned Crossed 8-bit Multiply)

Type: SIMD

Sub-extension: Zpsfoperand

Format:

UMUL8

31    25	24    20	19    15	14    12	11    7	6    0
UMUL8 1011100	Rs2	Rs1	000	Rd	OP-P 1110111

UMULX8

31    25	24    20	19    15	14    12	11    7	6    0
UMULX8 1011101	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

UMUL8 Rd, Rs1, Rs2  
UMULX8 Rd, Rs1, Rs2

Purpose: Perform unsigned 8-bit multiplications and generate four 16-bit results in parallel.

**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:

```
d_L = CONCAT(Rd(4,1),1'b0); d_H = CONCAT(Rd(4,1),1'b1);
// UMUL8
R[d_L].H[0] = Rs1.B[0] u* Rs2.B[0];
R[d_L].H[1] = Rs1.B[1] u* Rs2.B[1];
R[d_H].H[0] = Rs1.B[2] u* Rs2.B[2];
R[d_H].H[1] = Rs1.B[3] u* Rs2.B[3];
// UMULX8
R[d_L].H[0] = Rs1.B[0] u* Rs2.B[1];
R[d_L].H[1] = Rs1.B[1] u* Rs2.B[0];
R[d_H].H[0] = Rs1.B[2] u* Rs2.B[3];
R[d_H].H[1] = Rs1.B[3] u* Rs2.B[2];
```

- RV64:

```
// UMUL8
Rd.W[0].H[0] = Rs1.B[0] u* Rs2.B[0];
Rd.W[0].H[1] = Rs1.B[1] u* Rs2.B[1];
Rd.W[1].H[0] = Rs1.B[2] u* Rs2.B[2];
Rd.W[1].H[1] = Rs1.B[3] u* Rs2.B[3];
// UMULX8
Rd.W[0].H[0] = Rs1.B[0] u* Rs2.B[1];
Rd.W[0].H[1] = Rs1.B[1] u* Rs2.B[0];
Rd.W[1].H[0] = Rs1.B[2] u* Rs2.B[3];
Rd.W[1].H[1] = Rs1.B[3] u* Rs2.B[2];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- UMUL8
- Required:

```
uint64_t __rv_umul8(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint16x4_t __rv_v_umul8(uint8x4_t a, uint8x4_t b);
```

- UMULX8

- Required:

```
uint64_t __rv_umulx8(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint16x4_t __rv_v_umulx8(uint8x4_t a, uint8x4_t b);
```

## 103. UMUL16, UMULX16

### UMUL16 (SIMD Unsigned 16-bit Multiply)

#### UMULX16 (SIMD Unsigned Crossed 16-bit Multiply)

Type: SIMD

Sub-extension: Zpsfoperand

Format:

UMUL16

31    25	24    20	19    15	14    12	11    7	6    0
UMUL16 1011000	Rs2	Rs1	000	Rd	OP-P 1110111

UMULX16

31    25	24    20	19    15	14    12	11    7	6    0
UMULX16 1011001	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

UMUL16 Rd, Rs1, Rs2  
UMULX16 Rd, Rs1, Rs2

Purpose: Perform unsigned 16-bit multiplications and generate two 32-bit results in parallel.

**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:

```
d_L = CONCAT(Rd(4,1),1'b0); d_H = CONCAT(Rd(4,1),1'b1);
// UMUL16
R[d_H] = Rs1.H[1] u* Rs2.H[1];
R[d_L] = Rs1.H[0] u* Rs2.H[0];
// UMULX16
R[d_H] = Rs1.H[1] u* Rs2.H[0];
R[d_L] = Rs1.H[0] u* Rs2.H[1];
```

- RV64:

```
// UMUL16
Rd.W[1] = Rs1.H[1] u* Rs2.H[1];
Rd.W[0] = Rs1.H[0] u* Rs2.H[0];
// UMULX16
Rd.W[1] = Rs1.H[1] u* Rs2.H[0];
Rd.W[0] = Rs1.H[0] u* Rs2.H[1];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- UMUL16
- Required:

```
uint64_t __rv_umul16(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_umul16(uint16x2_t a, uint16x2_t b);
```

- UMULX16
- Required:

```
uint64_t __rv_umulx16(uint32_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_umulx16(uint16x2_t a, uint16x2_t b);
```

## 104. URADD64 (64-bit Unsigned Averaging Addition)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31	25	24	20	19	15	14	12	11	7	6	0
URADD64 1010000		Rs2		Rs1		001		Rd		OP-P 1110111	

Syntax:

```
URADD64 Rd, Rs1, Rs2
```

Purpose: Add two 64-bit unsigned integers. The result is averaged 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 65-bit addition result is first 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  $2d+1$ .

**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:

```
d_L = CONCAT(Rd(4,1),1'b0); d_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);
res65 = (ZE65(R[a_H].R[a_L]) + ZE65(R[b_H].R[b_L])) u>> 1;
R[d_H].R[d_L] = res65.D[0];
```

- RV64:

```
res65 = (ZE65(Rs1) + ZE65(Rs2)) u>> 1;
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint64_t __rv_uradd64(uint64_t a, uint64_t b);
```

## 105. URCRAS16 (SIMD 16-bit Unsigned Averaging Cross Addition & Subtraction)

Type: SIMD

Format:

31      25	24      20	19      15	14      12	11      7	6      0
URCRAS16 0010010	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
URCRAS16 Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk in parallel. Operands are from crossed positions in 32-bit chunks. The results are averaged 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 17-bit element results are first 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:

```
res_add17[x] = (ZE17(Rs1.W[x].H[1]) + ZE17(Rs2.W[x].H[0])) u>> 1;  
res_sub17[x] = (ZE17(Rs1.W[x].H[0]) - ZE17(Rs2.W[x].H[1])) u>> 1;  
Rd.W[x].H[1] = res_add17[x].H[0];  
Rd.W[x].H[0] = res_sub17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.H" and "PASUBU.H" instructions.

Intrinsic functions:

- Required:

```
uintXLEN_t _rv_urcras16(uintXLEN_t a, uintXLEN_t b);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_urcras16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_urcras16(uint16x4_t a, uint16x4_t b);
```

## 106. URCRSA16 (SIMD 16-bit Unsigned Averaging Cross Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
URCRSA16 0010011	Rs2	Rs1	000	Rd	OP-P 1110111

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk in parallel. Operands are from crossed positions in 32-bit chunks. The results are averaged 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 17-bit results are first 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:

```
res_sub17[x] = (ZE17(Rs1.W[x].H[1]) - ZE17(Rs2.W[x].H[0])) u>> 1;  
res_add17[x] = (ZE17(Rs1.W[x].H[0]) + ZE17(Rs2.W[x].H[1])) u>> 1;  
Rd.W[x].H[1] = res_sub17[x].H[0];  
Rd.W[x].H[0] = res_add17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.H" and "PASUBU.H" instructions.

Intrinsic functions:

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_urcrsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t __rv_v_urcrsa16(uint16x4_t a, uint16x4_t b);
```

## 107. URSTAS16 (SIMD 16-bit Unsigned Averaging Straight Addition & Subtraction)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
URSTAS16 1101010	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
URSTAS16 Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit unsigned integer element addition and 16-bit unsigned integer element subtraction in a 32-bit chunk in parallel. Operands are from corresponding positions in 32-bit chunks. The results are averaged 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 17-bit element results are first 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:

```
res_add17[x] = (ZE17(Rs1.W[x].H[1]) + ZE17(Rs2.W[x].H[1])) u>> 1;  
res_sub17[x] = (ZE17(Rs1.W[x].H[0]) - ZE17(Rs2.W[x].H[0])) u>> 1;  
Rd.W[x].H[1] = res_add17[x].H[0];  
Rd.W[x].H[0] = res_sub17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.H" and "PASUBU.H" instructions.

Intrinsic functions:

- Required:
- 
- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_urstas16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_urstas16(uint16x4_t a, uint16x4_t b);
```

## 108. URSTSA16 (SIMD 16-bit Unsigned Averaging Straight Subtraction & Addition)

Type: SIMD

Format:

31    25	24    20	19    15	14    12	11    7	6    0
URSTSA16 11010111	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
URCRSA16 Rd, Rs1, Rs2
```

**Purpose:** Perform 16-bit unsigned integer element subtraction and 16-bit unsigned integer element addition in a 32-bit chunk in parallel. Operands are from corresponding positions in 32-bit chunks. The results are averaged 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 17-bit results are first 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:

```
res_sub17[x] = (ZE17(Rs1.W[x].H[1]) - ZE17(Rs2.W[x].H[1])) u>> 1;  
res_add17[x] = (ZE17(Rs1.W[x].H[0]) + ZE17(Rs2.W[x].H[0])) u>> 1;  
Rd.W[x].H[1] = res_sub17[x].H[0];  
Rd.W[x].H[0] = res_add17[x].H[0];  
for RV32, x=0  
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.H" and "PASUBU.H" instructions.

Intrinsic functions:

- Required:
- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t _rv_v_urstsa16(uint16x2_t a, uint16x2_t b);
```

RV64:

```
uint16x4_t _rv_v_urstsa16(uint16x4_t a, uint16x4_t b);
```

## 109. URSUB64 (64-bit Unsigned Averaging Subtraction)

Type: RV32 and RV64

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
URSUB64 1010001	Rs2	Rs1	001	Rd	OP-P 1110111

Syntax:

```
URSUB64 Rd, Rs1, Rs2
```

Purpose: Perform a 64-bit unsigned integer subtraction. The result is averaged 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 65-bit subtraction result is first 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  $2d+1$ .

**RV64 Description:** This instruction subtracts the 64-bit unsigned integer in Rs2 from the 64-bit unsigned integer in Rs1. The 65-bit subtraction result is first right-shifted by 1 bit and then written to Rd.

Operations:

- RV32:

```
d_L = CONCAT(Rd(4,1),1'b0); d_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);
res65 = (ZE65(R[a_H].R[a_L]) - ZE65(R[b_H].R[b_L])) u>> 1;
R[d_H].R[d_L] = res65.D[0];
```

- RV64:

```
res65 = (ZE65(Rs1) - ZE65(Rs2)) u>> 1;
Rd = res65.D[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
uint64_t __rv_ursub64(uint64_t a, uint64_t b);
```

## 110. WEXTI (Extract Word from 64-bit Immediate)

Type: DSP

RV32: Replaced with FSRI in Zbpbo extension.

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
WEXTI 1101111	imm5u	Rs1	000	Rd	OP-P 1110111

Syntax:

```
WEXTI Rd, Rs1, imm5u
```

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, imm5u. 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, imm5u. The extracted word is sign-extended to 64-bits and written to Rd.

Operations:

- RV32:

```
Idx0 = CONCAT(Rs1(4,1),1'b0); Idx1 = CONCAT(Rs1(4,1),1'b1);
src[63:0] = Concat(R[Idx1], R[Idx0]);
LSB = imm5u;
Rd = src[31+LSB:LSB];
```

- RV64:

```
LSB = imm5u;  
ExtractW = Rs1[31+LSB:LSB];  
Rd = SE64(ExtractW)
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
intXLEN_t __rv_wext(uint64_t a, uint32_t b);
```

## 111. WEXT (Extract Word from 64-bit)

Type: DSP

RV32: Replaced with FSR in Zbpbo extension. RV64: Replaced with FSRW in Zbpbo extension.

Sub-extension: Zpsfoperand

Format:

31    25	24    20	19    15	14    12	11    7	6    0
WEXT 1100111	Rs2	Rs1	000	Rd	OP-P 1110111

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.

RV64 Description:

This instruction extracts a 32-bit word from a 64-bit value in Rs1 starting from a specified LSB bit position, specified in Rs2[4:0]. The extracted word is sign-extended to 64-bits and written to Rd.

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 = SE64(ExtractW)
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

```
intXLEN_t __rv_wext(uint64_t a, uint32_t b);
```

## 112. ZUNPKD810, ZUNPKD820, ZUNPKD830, ZUNPKD831, ZUNPKD832

ZUNPKD810 (Unsigned Unpacking Bytes 1 & 0)

ZUNPKD820 (Unsigned Unpacking Bytes 2 & 0)

ZUNPKD830 (Unsigned Unpacking Bytes 3 & 0)

ZUNPKD831 (Unsigned Unpacking Bytes 3 & 1)

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	OP-P 1110111

<u>xv</u>	<u>code[4:0]</u>
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**
- Required:

```
uintXLEN_t __rv_zunpkd810(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_zunpkd810(uint8x4_t a);
```

RV64:

```
uint16x4_t __rv_v_zunpkd810(uint8x8_t a);
```

- **ZUNPK820**
- Required:

```
uintXLEN_t __rv_zunpkd820(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_zunpkd820(uint8x4_t a);
```

RV64:

```
uint16x4_t __rv_v_zunpkd820(uint8x8_t a);
```

- **ZUNPK830**
- Required:

```
uintXLEN_t __rv_zunpkd830(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_zunpkd830(uint8x4_t a);
```

RV64:

```
uint16x4_t __rv_v_zunpkd830(uint8x8_t a);
```

- ZUNPK831

- Required:

```
uintXLEN_t __rv_zunpkd831(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_zunpkd831(uint8x4_t a);
```

RV64:

```
uint16x4_t __rv_v_zunpkd831(uint8x8_t a);
```

- ZUNPK832

- Required:

```
uintXLEN_t __rv_zunpkd832(uintXLEN_t a);
```

- Optional (e.g., GCC vector extensions):

RV32:

```
uint16x2_t __rv_v_zunpkd832(uint8x4_t a);
```

RV64:

```
uint16x4_t __rv_v_zunpkd832(uint8x8_t a);
```

## 113. KDMBB16, KDMBT16, KDMTT16

### KDMBB16 (SIMD Signed Saturating Double Multiply B16 x B16)

**KDMBT16 (SIMD Signed Saturating Double Multiply B16 x T16)**

**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	OP-P 1110111

#### KDMBT16

31    25	24    20	19    15	14    12	11    7	6    0
KDMBT16 1110101	Rs2	Rs1	001	Rd	OP-P 1110111

#### KDMTT16

31    25	24    20	19    15	14    12	11    7	6    0
KDMTT16 1111101	Rs2	Rs1	001	Rd	OP-P 1110111

**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 0xFFFFFFFF 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])) {
    Mres32[z] = aop[z] * bop[z];
    shifted33[z] = SE33(Mres32[z]) << 1;
    resQ31[z] = shifted33[z].W[0];
} else {
    resQ31[z] = 0x7FFFFFFF;
    OV = 1;
}
Rd.W[z] = resQ31[z];

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMBB16
- Required:

```
int64_t __rv_kdmbb16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kdmbb16(int16x4_t a, int16x4_t b);
```

- KDMBT16
- Required:

```
int64_t __rv_kdmkt16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kdmkt16(int16x4_t a, int16x4_t b);
```

- KDMTT16
- Required:

```
int64_t __rv_kdmkt16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t _rv_v_kdmtt16(int16x4_t a, int16x4_t b);
```

## 114. KDMABB16, KDMABT16, KDMATT16

### KDMABB16 (SIMD Signed Saturating Double Multiply Addition B16 x B16)

**KDMABT16 (SIMD Signed Saturating Double Multiply Addition B16 x T16)**

**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	OP-P 1110111

**KDMABT16**

31    25	24    20	19    15	14    12	11    7	6    0
KDMABT16 1110100	Rs2	Rs1	001	Rd	OP-P 1110111

**KDMATT16**

31    25	24    20	19    15	14    12	11    7	6    0
KDMATT16 1111100	Rs2	Rs1	001	Rd	OP-P 1110111

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 exceed 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])) {
    Mres32[z] = aop[z] * bop[z];
    shifted33[z] = SE33(Mres32[z]) << 1;
    resQ31[z] = shifted33[z].W[0];
} else {
    resQ31[z] = 0x7FFFFFFF;
    OV = 1;
}
resadd[z] = SE33(Rd.W[z]) + SE33(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].W[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KDMABB16
- Required:

```
int64_t _rv_kdmabb16(int64_t t, uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kdmabb16(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KDMABT16
- Required:

```
int64_t _rv_kdmabt16(int64_t t, uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kdmabt16(int32x2_t t, int16x4_t a, int16x4_t b);
```

- KDMATT16

- Required:

```
int64_t __rv_kdmatt16(int64_t t, uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kdmatt16(int32x2_t t, int16x4_t a, int16x4_t b);
```

## 115. KHMBB16, KHMBT16, KHMTT16

### KHMBB16 (SIMD Signed Saturating Half Multiply B16 x B16)

**KHMBT16 (SIMD Signed Saturating Half Multiply B16 x T16)**

**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	OP-P 1110111

**KHMBT16**

31    25	24    20	19    15	14    12	11    7	6    0
KHMBT16 1110110	Rs2	Rs1	001	Rd	OP-P 1110111

**KHMTT16**

31    25	24    20	19    15	14    12	11    7	6    0
KHMTT16 1111110	Rs2	Rs1	001	Rd	OP-P 1110111

**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)
// KHMTT16: (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] = 0xFFFF;
    OV = 1;
}
Rd.W[z] = SE32(res[15:0]);

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KHMBB16
- Required:

```
int64_t __rv_khmbb16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_khmbb16(int16x4_t a, int16x4_t b);
```

- KHMBT16
- Required:

```
int64_t __rv_khmbt16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_khmbt16(int16x4_t a, int16x4_t b);
```

- KHMTT16
- Required:

```
int64_t __rv_khmtt16(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_khmtt16(int16x4_t a, int16x4_t b);
```

## 116. KMSDA32, KMSXDA32

KMSDA32 (Saturating Signed Multiply Two Words & Add & Subtract)

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	OP-P 1110111

KMSXDA32

31 25	24 20	19 15	14 12	11 7	6 0
KMSXDA32 0100111	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

KMSDA32 Rd, Rs1, Rs2  
KMSXDA32 Rd, Rs1, Rs2

Purpose: Perform 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 exceeds 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:

```

mula64 = Rs1.W[1] s* Rs2.W[1]; mulb64 = Rs1.W[0] s* Rs2.W[0]; // KMSDA32
mula64 = Rs1.W[1] s* Rs2.W[0]; mulb64 = Rs1.W[0] s* Rs2.W[1]; // KMSXDA32
res66 = SE66(Rd) - SE66(mula64) - SE66(mulb64);
if (res66 > (2^63)-1) {
    res66 = (2^63)-1;
    OV = 1;
} else if (res66 < -2^63) {
    res66 = -2^63;
    OV = 1;
}
Rd = res66.D[0];

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KMSDA32
- Required:

```
int64_t __rv_kmsda32(int64_t t, int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int64_t __rv_v_kmsda32(int64_t t, int32x2_t a, int32x2_t b);
```

- KMSXDA32
- Required:

```
int64_t __rv_kmsxda32(int64_t t, int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int64_t __rv_v_kmsxda32(int64_t t, int32x2_t a, int32x2_t b);
```

# 117. 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		OP-P 1110111	

Syntax:

```
KSLL32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit elements logical left shift operations with saturation in parallel. 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.W[0];
} else {
    Rd = Rs1;
}
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t _rv_ksll32(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t _rv_v_ksll32(int32x2_t a, uint32_t b);
```

## 118. 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		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
KSLLI32 Rd, Rs1, imm5u
```

Purpose: Perform 32-bit elements logical left shift operations with saturation in parallel. 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;
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.W[0];
} else {
    Rd = Rs1;
}
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_ksll32(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_ksll32(int32x2_t a, uint32_t b);
```

## 119. KSLRA32, KSLRA32.u

KSLRA32 (SIMD 32-bit Shift Left Logical with Saturation or Shift Right Arithmetic)

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	OP-P 1110111

KSLRA32.u

31    25	24    20	19    15	14    12	11    7	6    0
KSLRA32.u 0110011	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
KSLRA32 Rd, Rs1, Rs2
KSLRA32.u Rd, Rs1, Rs2
```

**Purpose:** Perform 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]==-2<sup>5</sup> (0x20)" is defined to be equivalent to the behavior of "Rs2[5:0]==-(2<sup>5</sup>-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] u<< 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.W[0];
}
for RV64: x=1..0

```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- KSLRA32
- Required:

```
int64_t __rv_kslra32(int64_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kslra32(int32x2_t a, int32_t b);
```

- KSLRA32.u
- Required:

```
int64_t __rv_kslra32_u(int64_t a, int32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kslra32_u(int32x2_t a, int32_t b);
```

## 120. 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		OP-P 1110111	

Syntax:

```
KSTAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element saturating addition and 32-bit signed integer element saturating subtraction in a 64-bit chunk in parallel. 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 exceed 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] = SE33(Rs1.W[1]) + SE33(Rs2.W[1]);
res[0] = SE33(Rs1.W[0]) - SE33(Rs2.W[0]);
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[1] = res[1].W[0];
Rd.W[0] = res[0].W[0];
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_kstas32(int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kstas32(int32x2_t a, int32x2_t b);
```

# 121. KSTSA32 (SIMD 32-bit Signed Saturating Straight Subtraction & Addition)

Type: SIMD (RV64 Only)

Format:

31	25	24	20	19	15	14	12	11	7	6	0
KSTSA32 1100001		Rs2		Rs1		010		Rd		OP-P 1110111	

Syntax:

```
KSTSA32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit signed integer element saturating subtraction and 32-bit signed integer element saturating addition in a 64-bit chunk in parallel. 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 exceed 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] = SE33(Rs1.W[1]) - SE33(Rs2.W[1]);
res[0] = SE33(Rs1.W[0]) + SE33(Rs2.W[0]);
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[1] = res[1].W[0];
Rd.W[0] = res[0].W[0];
for RV64, x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
int64_t __rv_kstsa32(int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_kstsa32(int32x2_t a, int32x2_t b);
```

## 122. PKBB32, PKBT32, PKTT32, PKTB32

PKBB32 (Pack Two 32-bit Data from Both Bottom Half)

PKBT32 (Pack Two 32-bit Data from Bottom and Top Half)

PKTT32 (Pack Two 32-bit Data from Both Top Half)

Type: DSP (RV64 Only)

- PKBB32: RV64: Replaced with PACK in Zbpbo
- PKTT32: RV64: Replaced with PACKU in Zbpbo

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	OP-P 1110111

<u>xv</u>	<u>zz</u>
BT	01
TB	11

For RV64 PACK/PACKU encoding format, please see Section 6.7.

Syntax:

```
PKBB32 Rd, Rs1, Rs2 == PACK Rd, Rs2, Rs1
PKBT32 Rd, Rs1, Rs2
PKTT32 Rd, Rs1, Rs2 == PACKU Rd, Rs2, Rs1
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
- Required:

```
uint64_t __rv_pkbb32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_pkbb32(uint32x2_t a, uint32x2_t b);
```

- PKBT32
- Required:

```
uint64_t __rv_pkbt32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_pkbt32(uint32x2_t a, uint32x2_t b);
```

- PKTB32
- Required:

```
uint64_t __rv_pktb32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_pktb32(uint32x2_t a, uint32x2_t b);
```

- PKTT32

- Required:

```
uint64_t __rv_pktt32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_pktt32(uint32x2_t a, uint32x2_t b);
```

# 123. RSTAS32 (SIMD 32-bit Signed Averaging 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	OP-P 1110111

Syntax:

```
RSTAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit signed integer element addition and 32-bit signed integer element subtraction in a 64-bit chunk in parallel. Operands are from corresponding 32-bit elements. The results are averaged 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:

```
res_add33 = (SE33(Rs1.W[1]) + SE33(Rs2.W[1])) s>> 1;  
res_sub33 = (SE33(Rs1.W[0]) - SE33(Rs2.W[0])) s>> 1;  
Rd.W[1] = res_add33.W[0];  
Rd.W[0] = res_sub33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.W" and "PASUB.W" instructions.

Intrinsic functions:

- Required:

```
int64_t __rv_rstas32(int64_t a, int64_t b);
```

```
int32x2_t __rv_v_rstas32(int32x2_t a, int32x2_t b);
```

## 124. RSTSA32 (SIMD 32-bit Signed Averaging 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		OP-P 1110111	

Syntax:

```
RSTSA32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit signed integer element subtraction and 32-bit signed integer element addition in a 64-bit chunk in parallel. Operands are from corresponding 32-bit elements. The results are averaged 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:

```
res_sub33 = (SE33(Rs1.W[1]) - SE33(Rs2.W[1])) s>> 1;  
res_add33 = (SE33(Rs1.W[0]) + SE33(Rs2.W[0])) s>> 1;  
Rd.W[1] = res_sub33.W[0];  
Rd.W[0] = res_add33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADD.W" and "PASUB.W" instructions.

Intrinsic functions:

- Required:

```
int64_t __rv_rstsa32(int64_t a, int64_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_rstsa32(int32x2_t a, int32x2_t b);
```

## 125. 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	OP-P 1110111

Syntax:

```
SLL32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit elements logical left shift operations in parallel. 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:

- Required:

```
uint64_t __rv_sll32(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_sll32(uint32x2_t a, uint32_t b);
```

## 126. 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	Rs1	010	Rd	OP-P 1110111

Syntax:

```
SLLI32 Rd, Rs1, imm5u
```

Purpose: Perform 32-bit element logical left shift operations in parallel. 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 constant. And the results are written to Rd.

Operations:

```
sa = imm5u;  
Rd.W[x] = Rs1.W[x] << sa;  
for RV64: x=1..0
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_sll32(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_sll32(uint32x2_t a, uint32_t b);
```

## 127. SRA32, SRA32.u

SRA32 (SIMD 32-bit Shift Right Arithmetic)

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	OP-P 1110111

### SRA32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRA32.u 0110000	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
SRA32 Rd, Rs1, Rs2
SRA32.u Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit element arithmetic right shift operations in parallel. 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
- Required:

```
int64_t __rv_sra32(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_sra32(int32x2_t a, uint32_t b);
```

- SRA32.u
- Required:

```
int64_t __rv_sra32_u(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):



## 128. SRAI32, SRAI32.u

SRAI32 (SIMD 32-bit Shift Right Arithmetic Immediate)

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	Rs1	010	Rd	OP-P 1110111

SRAI32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRAI32u 1000000	imm5u	Rs1	010	Rd	OP-P 1110111

Syntax:

SRAI32 Rd, Rs1, imm5u  
SRAI32.u Rd, Rs1, imm5u

Purpose: Perform 32-bit elements arithmetic right shift operations in parallel. 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;
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
- Required:

```
int64_t __rv_sra32(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
int32x2_t __rv_v_sra32(int32x2_t a, uint32_t b);
```

- SRAI32.u
- Required:

```
int64_t __rv_sra32_u(int64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

## 129. 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	Rs1	001	Rd	OP-P 1110111

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:

```
int32_t __rv_sraw_u(int32_t a, uint32_t b);
```

## 130. SRL32, SRL32.u

### SRL32 (SIMD 32-bit Shift Right Logical)

#### 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	OP-P 1110111

SRL32.u

31    25	24    20	19    15	14    12	11    7	6    0
SRL32.u 0110001	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
SRL32 Rd, Rs1, Rs2
SRL32.u Rd, Rs1, Rs2
```

Purpose: Perform 32-bit element logical right shift operations in parallel. 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) { // SRL32.u
        res[31:-1] = ZE33(Rs1.W[x][31:sa-1]) + 1;
        Rd.W[x] = res[31:0];
    } else { // SRL32
        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
- Required:

```
uint64_t __rv_srl32(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_srl32(uint32x2_t a, uint32_t b);
```

- SRL32.u
- Required:

```
uint64_t __rv_srl32_u(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

## 131. SRLI32, SRLI32.u

SRLI32 (SIMD 32-bit Shift Right Logical Immediate)

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	Rs1	010	Rd	OP-P 1110111

SRLI32.u

31 25	24 20	19 15	14 12	11 7	6 0
SRLI32u 1000001	imm5u	Rs1	010	Rd	OP-P 1110111

Syntax:

SRLI32 Rd, Rs1, imm5u

SRLI32.u Rd, Rs1, imm5u

**Purpose:** Perform 32-bit elements logical right shift operations in parallel. 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;
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
- Required:

```
uint64_t __rv_srl32(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_srl32(uint32x2_t a, uint32_t b);
```

- SRLI32.u
- Required:

```
uint64_t __rv_srl32_u(uint64_t a, uint32_t b);
```

- Optional (e.g., GCC vector extensions):

## 132. 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		OP-P 1110111	

Syntax:

```
STAS32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit integer element addition and 32-bit integer element subtraction in a 64-bit chunk in parallel. 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:

- Required:

```
uint64_t __rv_stas32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ustas32(uint32x2_t a, uint32x2_t b);  
int32x2_t __rv_v_sstas32(int32x2_t a, int32x2_t b);
```

# 133. 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		OP-P 1110111	

Syntax:

```
STSA32 Rd, Rs1, Rs2
```

Purpose: Perform 32-bit integer element subtraction and 32-bit integer element addition in a 64-bit chunk in parallel. 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:

- Required:

```
uint64_t __rv_stsa32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ustsa32(uint32x2_t a, uint32x2_t b);  
int32x2_t __rv_v_ssts32(int32x2_t a, int32x2_t b);
```

## 134. 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	OP-P 1110111

Syntax:

```
UKCRAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk in parallel. 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 exceed 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 = ZE33(Rs1.W[1]) + ZE33(Rs2.W[0]);
res2 = ZE33(Rs1.W[0]) - ZE33(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.W[0];
Rd.W[0] = res2.W[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_ukcras32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ukcras32(uint32x2_t a, uint32x2_t b);
```

## 135. 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
UKCRSA32 0011011	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
UKCRSA32 Rd, Rs1, Rs2
```

**Purpose:** Perform one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk in parallel. 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] of Rs1. If any of the results exceed 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 = ZE33(Rs1.W[1]) - ZE33(Rs2.W[0]);
res2 = ZE33(Rs1.W[0]) + ZE33(Rs2.W[1]);
if (res1 < 0) {
    res1 = 0;
    OV = 1;
}
if (res2 > (2^32)-1) {
    res2 = (2^32)-1;
    OV = 1;
}
Rd.W[1] = res1.W[0];
Rd.W[0] = res2.W[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_ukcrsa32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ukcrsa32(uint32x2_t a, uint32x2_t b);
```

## 136. 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		OP-P 1110111	

Syntax:

```
UKSTAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform one 32-bit unsigned integer element saturating addition and one 32-bit unsigned integer element saturating subtraction in a 64-bit chunk in parallel. 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 exceed 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 = ZE33(Rs1.W[1]) + ZE33(Rs2.W[1]);
res2 = ZE33(Rs1.W[0]) - ZE33(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.W[0];
Rd.W[0] = res2.W[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_ukstas32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ukstas32(uint32x2_t a, uint32x2_t b);
```

## 137. 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		OP-P 1110111	

Syntax:

```
UKSTSA32 Rd, Rs1, Rs2
```

**Purpose:** Perform one 32-bit unsigned integer element saturating subtraction and one 32-bit unsigned integer element saturating addition in a 64-bit chunk in parallel. 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 exceed 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 = ZE33(Rs1.W[1]) - ZE33(Rs2.W[1]);
res2 = ZE33(Rs1.W[0]) + ZE33(Rs2.W[0]);
if (res1 < 0) {
    res1 = 0;
    OV = 1;
}
if (res2 > (2^32)-1) {
    res2 = (2^32)-1;
    OV = 1;
}
Rd.W[1] = res1.W[0];
Rd.W[0] = res2.W[0];
```

Exceptions: None

Privilege level: All

Note:

Intrinsic functions:

- Required:

```
uint64_t __rv_ukstsa32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t __rv_v_ukstsa32(uint32x2_t a, uint32x2_t b);
```

## 138. URCRAS32 (SIMD 32-bit Unsigned Averaging 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	OP-P 1110111

Syntax:

```
URCRAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk in parallel. Operands are from crossed 32-bit elements. The results are averaged 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 33-bit element results are first right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res_add33 = (ZE33(Rs1.W[1]) + ZE33(Rs2.W[0])) u>> 1;  
res_sub33 = (ZE33(Rs1.W[0]) - ZE33(Rs2.W[1])) u>> 1;  
Rd.W[1] = res_add33.W[0];  
Rd.W[0] = res_sub33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.W" and "PASUBU.W" instructions.

Intrinsic functions:

- Required:

```
uint64_t _rv_urcras32(uint64_t a, uint64_t b);
```

- Optional (e.g., GCC vector extensions):

```
uint32x2_t _rv_v_urcras32(uint32x2_t a, uint32x2_t b);
```

# 139. URCRSA32 (SIMD 32-bit Unsigned Averaging 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	OP-P 1110111

Syntax:

```
URCRSA32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk in parallel. Operands are from crossed 32-bit elements. The results are averaged 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 33-bit results are first right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res_sub33 = (ZE33(Rs1.W[1]) - ZE33(Rs2.W[0])) u>> 1;  
res_add33 = (ZE33(Rs1.W[0]) + ZE33(Rs2.W[1])) u>> 1;  
Rd.W[1] = res_sub33.W[0];  
Rd.W[0] = res_add33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.W" and "PASUBU.W" instructions.

Intrinsic functions:

- Required:

```
uint32x2_t _rv_v_urcrsa32(uint32x2_t a, uint32x2_t b);
```

- Optional (e.g., GCC vector extensions):

## 140. URSTAS32 (SIMD 32-bit Unsigned Averaging Straight Addition & Subtraction)

Type: SIMD (RV64 Only)

Format:

31    25	24    20	19    15	14    12	11    7	6    0
URSTAS32 1101000	Rs2	Rs1	010	Rd	OP-P 1110111

Syntax:

```
URSTAS32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit unsigned integer element addition and 32-bit unsigned integer element subtraction in a 64-bit chunk in parallel. Operands are from corresponding 32-bit elements. The results are averaged 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 33-bit element results are first right-shifted by 1 bit and then written to [63:32] of Rd for addition and [31:0] of Rd for subtraction.

Operations:

```
res_add33 = (ZE33(Rs1.W[1]) + ZE33(Rs2.W[1])) u>> 1;  
res_sub33 = (ZE33(Rs1.W[0]) - ZE33(Rs2.W[0])) u>> 1;  
Rd.W[1] = res_add33.W[0];  
Rd.W[0] = res_sub33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.W" and "PASUBU.W" instructions.

Intrinsic functions:

- Required:

```
uint32x2_t _rv_v_urstas32(uint32x2_t a, uint32x2_t b);
```

- Optional (e.g., GCC vector extensions):

# 141. URSTSA32 (SIMD 32-bit Unsigned Averaging 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	OP-P 1110111

Syntax:

```
URSTSA32 Rd, Rs1, Rs2
```

**Purpose:** Perform 32-bit unsigned integer element subtraction and 32-bit unsigned integer element addition in a 64-bit chunk in parallel. Operands are from corresponding 32-bit elements. The results are averaged 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 33-bit results are first right-shifted by 1 bit and then written to [63:32] of Rd for subtraction and [31:0] of Rd for addition.

Operations:

```
res_sub33 = (ZE33(Rs1.W[1]) - ZE33(Rs2.W[1])) u>> 1;  
res_add33 = (ZE33(Rs1.W[0]) + ZE33(Rs2.W[0])) u>> 1;  
Rd.W[1] = res_sub33.W[0];  
Rd.W[0] = res_add33.W[0];
```

Exceptions: None

Privilege level: All

Examples: Please see "PAADDU.W" and "PASUBU.W" instructions.

Intrinsic functions:

- Required:

```
uint32x2_t _rv_v_urstsa32(uint32x2_t a, uint32x2_t b);
```

- Optional (e.g., GCC vector extensions):