The RISC-V Instruction Set Manual, Volume I

Unprivileged Architecture

Version 20260120: Official Release

Table of Contents

Preamble 1
Preface 2
1. Introduction 13
1.1. RISC-V Hardware Platform Terminology 14
1.2. RISC-V Software Execution Environments and Harts 14
1.3. RISC-V ISA Overview 15
1.4. Memory 17
1.5. Base Instruction-Length Encoding 18
1.6. Exceptions, Traps, and Interrupts 20
1.7. UNSPECIFIED Behaviors and Values 21
2. RV32I Base Integer Instruction Set, Version 2.1 23
2.1. Programmers' Model for Base Integer ISA 23
2.2. Base Instruction Formats 25
2.3. Immediate Encoding Variants 26
2.4. Integer Computational Instructions 27
2.4.1. Integer Register-Immediate Instructions 28
2.4.2. Integer Register-Register Instructions 29
2.4.3. NOP Instruction 29
2.5. Control Transfer Instructions 30
2.5.1. Unconditional Jumps 30
2.5.2. Conditional Branches 31
2.6. Load and Store Instructions 33
2.7. Memory Ordering Instructions 35
2.8. Environment Call and Breakpoints 36
2.9. HINT Instructions 37
3. RV32E and RV64E Base Integer Instruction Sets, Version 2.0 40
3.1. RV32E and RV64E Programmers' Model 40
3.2. RV32E and RV64E Instruction Set Encoding 40
4. RV64I Base Integer Instruction Set, Version 2.1 41
4.1. Register State 41
4.2. Integer Computational Instructions 41
4.2.1. Integer Register-Immediate Instructions 41
4.2.2. Integer Register-Register Operations 42
4.3. Load and Store Instructions 42
4.4. HINT Instructions 43
5. "Zifencei" Extension for Instruction-Fetch Fence, Version 2.0 45
6. "Zicsr" Extension for Control and Status Register (CSR) Instructions, Version 2.0 47
6.1. CSR Instructions 47
6.1.1. CSR Access Ordering 49
7. "Zicntr" and "Zihpm" Extensions for Counters, Version 2.0 51
7.1. "Zicntr" Extension for Base Counters and Timers 51
7.2. "Zihpm" Extension for Hardware Performance Counters 53
8. "Zihintntl" Extension for Non-Temporal Locality Hints, Version 1.0 55

9. "Zihintpause" Extension for Pause Hint, Version 2.0 59
10. "Zimop" Extension for May-Be-Operations, Version 1.0 60
10.1. "Zcmop" Compressed May-Be-Operations Extension, Version 1.0 61
11. "Zicond" Extension for Integer Conditional Operations, Version 1.0.0 62
11.1. Instructions (in alphabetical order) 62
11.1.1. czero.eqz 62
11.1.2. czero.nez 63
11.2. Usage examples 63
11.2.1. Instruction sequences 64
12. "M" Extension for Integer Multiplication and Division, Version 2.0 65
12.1. Multiplication Operations 65
12.2. Division Operations 65
12.3. `Zmmul` Extension, Version 1.0 66
13. "A" Extension for Atomic Instructions, Version 2.1 67
13.1. Specifying Ordering of Atomic Instructions 67
13.2. "Zalrsc" Extension for Load-Reserved/Store-Conditional Instructions 67
13.3. Eventual Success of Store-Conditional Instructions 70
13.4. "Zaamo" Extension for Atomic Memory Operations 72
14. "Zawrs" Extension for Wait-on-Reservation-Set instructions, Version 1.01 74
14.1. Wait-on-Reservation-Set Instructions 74
15. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions, Version 1.0.0 76
15.1. Word/Doubleword/Quadword CAS (AMOCAS.W/D/Q) Instructions 76
16. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0 81
16.1. Byte and Halfword Atomic Memory Operation Instructions 81
17. "Zalasr" Atomic Load-Acquire and Store-Release Instructions, Version 1.0 82
17.1. Load-Acquire and Store-Release Instructions 82
17.2. Load Acquire 83
17.3. Store Release 84
18. RVWMO Memory Consistency Model, Version 2.0 85
18.1. Definition of the RVWMO Memory Model 85
18.1.1. Memory Model Primitives 85
18.1.2. Syntactic Dependencies 87
18.1.3. Preserved Program Order 88
18.1.4. Memory Model Axioms 89
18.1.4.1. Load Value Axiom 89
18.1.4.2. Atomicity Axiom 89
18.1.4.3. Progress Axiom 89
18.2. CSR Dependency Tracking Granularity 89
18.3. Source and Destination Register Listings 89
19. "Ztso" Extension for Total Store Ordering, Version 1.0 96
20. "CMO" Extensions for Base Cache Management Operation ISA, Version 1.0.0 97
20.1. Pseudocode for instruction semantics 97
20.2. Introduction 97
20.3. Background 97
20.3.1. Memory and Caches 97

20.3.2. Cache-Block Operations 98
20.4. Coherent Agents and Caches 99
20.4.1. Memory Ordering 100
20.4.1.1. Preserved Program Order 100
20.4.1.2. Load Values 100
20.4.2. Traps 101
20.4.2.1. Illegal-Instruction and Virtual-Instruction Exceptions 101
20.4.2.2. Page-Fault, Guest-Page-Fault, and Access-Fault Exceptions 101
20.4.2.3. Address-Misaligned Exceptions 102
20.4.2.4. Breakpoint Exceptions and Debug Mode Entry 102
20.4.2.5. Hypervisor Extension 103
20.4.3. Effects on Constrained LR/SC Loops 104
20.4.4. Software Discovery 104
20.5. CSR controls for CMO instructions 104
20.6. Extensions 106
20.6.1. Cache-Block Management Instructions 106
20.6.2. Cache-Block Zero Instructions 107
20.6.3. Cache-Block Prefetch Instructions 107
20.7. Instructions 108
20.7.1. cbo.clean 108
20.7.2. cbo.flush 108
20.7.3. cbo.inval 109
20.7.4. cbo.zero 109
20.7.5. prefetch.i 110
20.7.6. prefetch.r 110
20.7.7. prefetch.w 111
21. "F" Extension for Single-Precision Floating-Point, Version 2.2 112
21.1. F Register State 112
21.2. Floating-Point Control and Status Register 113
21.3. NaN Generation and Propagation 114
21.4. Subnormal Arithmetic 115
21.5. Single-Precision Load and Store Instructions 115
21.6. Single-Precision Floating-Point Computational Instructions 116
21.7. Single-Precision Floating-Point Conversion and Move Instructions 117
21.8. Single-Precision Floating-Point Compare Instructions 119
21.9. Single-Precision Floating-Point Classify Instruction 119
22. "D" Extension for Double-Precision Floating-Point, Version 2.2 121
22.1. D Register State 121
22.2. NaN Boxing of Narrower Values 121
22.3. Double-Precision Load and Store Instructions 122
22.4. Double-Precision Floating-Point Computational Instructions 122
22.5. Double-Precision Floating-Point Conversion and Move Instructions 123
22.6. Double-Precision Floating-Point Compare Instructions 124
22.7. Double-Precision Floating-Point Classify Instruction 124
23. "Q" Extension for Quad-Precision Floating-Point, Version 2.2 125

23.1. Quad-Precision Load and Store Instructions ... 125
23.2. Quad-Precision Computational Instructions ... 125
23.3. Quad-Precision Convert and Move Instructions ... 126
23.4. Quad-Precision Floating-Point Compare Instructions ... 126
23.5. Quad-Precision Floating-Point Classify Instruction ... 127
24. "Zfh" and "Zfhmin" Extensions for Half-Precision Floating-Point, Version 1.0 ... 128
24.1. Half-Precision Load and Store Instructions ... 128
24.2. Half-Precision Computational Instructions ... 128
24.3. Half-Precision Conversion and Move Instructions ... 129
24.4. Half-Precision Floating-Point Compare Instructions ... 130
24.5. Half-Precision Floating-Point Classify Instruction ... 130
24.6. "Zfhmin" Standard Extension for Minimal Half-Precision Floating-Point ... 130
25. "BF16" Extensions for BFloat16-precision Floating-Point, Version 1.0 ... 132
25.1. Introduction ... 132
25.2. Intended Audience ... 132
25.3. Number Format ... 133
25.3.1. BF16 Operand Format ... 133
25.3.2. BF16 Behavior ... 134
25.3.2.1. Subnormal Numbers: ... 134
25.3.2.2. Infinities: ... 134
25.3.2.3. NaNs ... 134
25.3.2.4. Scalar NaN Boxing ... 135
25.3.2.5. Rounding Modes: ... 135
25.3.2.6. Handling exceptions ... 135
25.3.2.6.1. Underflow ... 135
25.4. Extensions ... 137
25.4.1. `Zfbfmin` - Scalar BF16 Converts ... 137
25.4.2. `Zvfbfmin` - Vector BF16 Converts ... 138
25.4.3. `Zvfbfwma` - Vector BF16 widening mul-add ... 138
25.5. Instructions ... 139
25.5.1. fcvt.bf16.s ... 139
25.5.2. fcvt.s.bf16 ... 140
25.5.3. vfncvtbf16.f.f.w ... 141
25.5.4. vfwcvtbf16.f.f.v ... 142
25.5.5. vfwmaccbf16 ... 143
Bibliography ... 144
26. "Zfa" Extension for Additional Floating-Point Instructions, Version 1.0 ... 145
26.1. Load-Immediate Instructions ... 145
26.2. Minimum and Maximum Instructions ... 146
26.3. Round-to-Integer Instructions ... 147
26.4. Modular Convert-to-Integer Instruction ... 147
26.5. Move Instructions ... 147
26.6. Comparison Instructions ... 148
27. "Zfinx", "Zdinx", "Zhinx", "Zhinxmin" Extensions for Floating-Point in Integer Registers, Version 1.0 ... 149

27.1. Processing of Narrower Values 149
27.2. Zdinx 149
27.3. Processing of Wider Values 150
27.4. Zhinx 150
27.5. Zhinxmin 150
27.6. Privileged Architecture Implications 150
28. "C" Extension for Compressed Instructions, Version 2.0 152
28.1. Overview 152
28.2. Compressed Instruction Formats 154
28.3. Load and Store Instructions 155
28.3.1. Stack-Pointer-Based Loads and Stores 155
28.3.2. Register-Based Loads and Stores 157
28.4. Control Transfer Instructions 158
28.5. Integer Computational Instructions 159
28.5.1. Integer Constant-Generation Instructions 159
28.5.2. Integer Register-Immediate Operations 159
28.5.3. Integer Register-Register Operations 161
28.5.4. Defined Illegal Instruction 162
28.5.5. NOP Instruction 162
28.5.6. Breakpoint Instruction 162
28.6. Usage of C Instructions in LR/SC Sequences 163
28.7. HINT Instructions 163
28.8. RVC Instruction Set Listings 164
29. "Zc*" Extension for Code Size Reduction, Version 1.0.0 167
29.1. Zc* Overview 167
29.2. C 168
29.3. Zce 168
29.4. MISA.C 168
29.5. Zca 169
29.6. Zcf (RV32 only) 169
29.7. Zcd 169
29.8. Zcb 169
29.9. Zcmp 171
29.10. Zcmt 172
29.11. Zc instruction formats 172
29.12. Zcb instructions 173
29.12.1. c.lbu 173
29.12.2. c.lhu 174
29.12.3. c.lh 175
29.12.4. c.sb 176
29.12.5. c.sh 177
29.12.6. c.zext.b 178
29.12.7. c.sext.b 179
29.12.8. c.zext.h 180
29.12.9. c.sext.h 181

29.12.10. c.zext.w 182
29.12.11. c.not 183
29.12.12. c.mul 184
29.13. PUSH/POP register instructions 185
29.13.1. PUSH/POP functional overview 185
29.13.2. Example usage 186
29.13.2.1. Stack pointer adjustment handling 187
29.13.2.2. Register list handling 187
29.13.3. PUSH/POP Fault handling 187
29.13.4. Software view of execution 188
29.13.4.1. Software view of the PUSH sequence 188
29.13.4.2. Software view of the POP/POPRET sequence 189
29.13.5. Non-idempotent memory handling 190
29.13.6. Example RV32I PUSH/POP sequences 191
29.13.6.1. cm.push \{ra, s0-s2}, -64 191
29.13.6.2. cm.push \{ra, s0-s11}, -112 191
29.13.6.3. cm.pop {ra}, 16 192
29.13.6.4. cm.pop \{ra, s0-s3}, 48 192
29.13.6.5. cm.pop \{ra, s0-s4}, 64 192
29.13.7. cm.push 193
29.13.8. cm.pop 198
29.13.9. cm.popretz 203
29.13.10. cm.popret 208
29.13.11. cm.mvsa01 213
29.13.12. cm.mva01s 214
29.14. Table Jump Overview 215
29.14.1. jvt 215
29.14.2. Table Jump Fault handling 215
29.14.3. jvt CSR 216
29.14.4. cm.jt 217
29.14.5. cm.jalt 219
30. Bit Manipulation Extensions 221
30.1. "B" Extension for Bit Manipulation, Version 1.0.0 223
30.2. Zba: Extension for Address generation, Version 1.0.0 223
30.3. Zbb: Extension for Basic bit-manipulation, Version 1.0.0 223
30.3.1. Logical with negate 223
30.3.2. Count leading/trailing zero bits 224
30.3.3. Count population 224
30.3.4. Integer minimum/maximum 224
30.3.5. Sign extension and zero extension 224
30.3.6. Bitwise rotation 225
30.3.7. OR Combine 225
30.3.8. Byte-reverse 225
30.4. Zbc: Extension for Carry-less multiplication, Version 1.0.0 225
30.5. Zbs: Extension for Single-bit instructions, Version 1.0.0 226

30.6. Zbkb: Extension for Bit-manipulation for Cryptography, Version 1.0.0 226
30.7. Zbkc: Extension for Carry-less multiplication for Cryptography, Version 1.0.0 227
30.8. Zbkx: Extension for Crossbar permutations, Version 1.0.0 227
30.9. Instructions (in alphabetical order) 228
30.9.1. add.uw 228
30.9.2. andn 229
30.9.3. bclr 230
30.9.4. bclri 231
30.9.5. bext 232
30.9.6. bexti 233
30.9.7. binv 234
30.9.8. binvi 235
30.9.9. bset 236
30.9.10. bseti 237
30.9.11. clmul 238
30.9.12. clmulh 239
30.9.13. clmulr 240
30.9.14. clz 241
30.9.15. clzw 242
30.9.16. cpop 243
30.9.17. cpopw 244
30.9.18. ctz 245
30.9.19. ctzw 246
30.9.20. max 247
30.9.21. maxu 248
30.9.22. min 249
30.9.23. minu 250
30.9.24. orc.b 251
30.9.25. orn 252
30.9.26. pack 253
30.9.27. packh 254
30.9.28. packw 255
30.9.29. rev8 256
30.9.30. brev8 257
30.9.31. rol 258
30.9.32. rolw 259
30.9.33. ror 260
30.9.34. rori 261
30.9.35. roriw 262
30.9.36. rorw 263
30.9.37. sext.b 264
30.9.38. sext.h 265
30.9.39. sh1add 266
30.9.40. sh1add.uw 267
30.9.41. sh2add 268

30.9.42. sh2add.uw ... 269
30.9.43. sh3add ... 270
30.9.44. sh3add.uw ... 271
30.9.45. slli.uw ... 272
30.9.46. unzip ... 273
30.9.47. xnor ... 274
30.9.48. xperm8 ... 275
30.9.49. xperm4 ... 276
30.9.50. zext.h ... 277
30.9.51. zip ... 278
31. "V" Standard Extension for Vector Operations, Version 1.0 ... 279
31.1. Introduction ... 279
31.2. Implementation-defined Constant Parameters ... 279
31.3. Vector Extension Programmer's Model ... 279
31.3.1. Vector Registers ... 280
31.3.2. Vector Context Status in `mstatus` ... 280
31.3.3. Vector Context Status in `vsstatus` ... 280
31.3.4. Vector Type (`vtype`) Register ... 281
31.3.4.1. Vector Selected Element Width (`vsew[2:0]`) ... 282
31.3.4.2. Vector Register Grouping (`vlmul[2:0]`) ... 282
31.3.4.3. Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma` ... 284
31.3.4.4. Vector Type Illegal (`vill`) ... 285
31.3.5. Vector Length (`vl`) Register ... 286
31.3.6. Vector Byte Length (`vlenb`) Register ... 286
31.3.7. Vector Start Index (`vstart`) Register ... 286
31.3.8. Vector Fixed-Point Rounding Mode (`vxrm`) Register ... 287
31.3.9. Vector Fixed-Point Saturation Flag (`vxsat`) ... 288
31.3.10. Vector Control and Status (`vcsr`) Register ... 288
31.3.11. State of Vector Extension at Reset ... 288
31.4. Mapping of Vector Elements to Vector Register State ... 288
31.4.1. Mapping for LMUL = 1 ... 289
31.4.2. Mapping for LMUL < 1 ... 290
31.4.3. Mapping for LMUL > 1 ... 290
31.4.4. Mapping across Mixed-Width Operations ... 291
31.4.5. Mask Register Layout ... 292
31.5. Vector Instruction Formats ... 292
31.5.1. Scalar Operands ... 294
31.5.2. Vector Operands ... 294
31.5.3. Vector Masking ... 295
31.5.3.1. Mask Encoding ... 296
31.5.4. Prestart, Active, Inactive, Body, and Tail Element Definitions ... 296
31.6. Configuration-Setting Instructions (`vsetvli`/`vsetivli`/`vsetvl`) ... 297
31.6.1. `vtype` encoding ... 298
31.6.1.1. Unsupported `vtype` Values ... 299
31.6.2. AVL encoding ... 299

31.6.3. Constraints on Setting `vl` 300
31.6.4. Example of strip mining and changes to SEW 301
31.7. Vector Loads and Stores 301
31.7.1. Vector Load/Store Instruction Encoding 302
31.7.2. Vector Load/Store Addressing Modes 303
31.7.3. Vector Load/Store Width Encoding 304
31.7.4. Vector Unit-Stride Instructions 305
31.7.5. Vector Constant-Stride Instructions 306
31.7.6. Vector Indexed Instructions 307
31.7.7. Unit-stride Fault-Only-First Loads 308
31.7.8. Vector Load/Store Segment Instructions 309
31.7.8.1. Vector Unit-Stride Segment Loads and Stores 310
31.7.8.2. Vector Constant-Stride Segment Loads and Stores 311
31.7.8.3. Vector Indexed Segment Loads and Stores 312
31.7.9. Vector Load/Store Whole Register Instructions 313
31.8. Vector Memory Alignment Constraints 315
31.9. Vector Memory Consistency Model 315
31.10. Vector Arithmetic Instruction Formats 316
31.10.1. Vector Arithmetic Instruction encoding 316
31.10.2. Widening Vector Arithmetic Instructions 318
31.10.3. Narrowing Vector Arithmetic Instructions 319
31.11. Vector Integer Arithmetic Instructions 320
31.11.1. Vector Single-Width Integer Add and Subtract 320
31.11.2. Vector Widening Integer Add/Subtract 320
31.11.3. Vector Integer Extension 321
31.11.4. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions 321
31.11.5. Vector Bitwise Logical Instructions 323
31.11.6. Vector Single-Width Shift Instructions 324
31.11.7. Vector Narrowing Integer Right Shift Instructions 324
31.11.8. Vector Integer Compare Instructions 325
31.11.9. Vector Integer Min/Max Instructions 328
31.11.10. Vector Single-Width Integer Multiply Instructions 328
31.11.11. Vector Integer Divide Instructions 329
31.11.12. Vector Widening Integer Multiply Instructions 329
31.11.13. Vector Single-Width Integer Multiply-Add Instructions 330
31.11.14. Vector Widening Integer Multiply-Add Instructions 330
31.11.15. Vector Integer Merge Instructions 331
31.11.16. Vector Integer Move Instructions 331
31.12. Vector Fixed-Point Arithmetic Instructions 332
31.12.1. Vector Single-Width Saturating Add and Subtract 332
31.12.2. Vector Single-Width Averaging Add and Subtract 332
31.12.3. Vector Single-Width Fractional Multiply with Rounding and Saturation 333
31.12.4. Vector Single-Width Scaling Shift Instructions 333
31.12.5. Vector Narrowing Fixed-Point Clip Instructions 334
31.13. Vector Floating-Point Instructions 335

31.13.1. Vector Floating-Point Exception Flags ... 335
31.13.2. Vector Single-Width Floating-Point Add/Subtract Instructions ... 335
31.13.3. Vector Widening Floating-Point Add/Subtract Instructions ... 336
31.13.4. Vector Single-Width Floating-Point Multiply/Divide Instructions ... 336
31.13.5. Vector Widening Floating-Point Multiply ... 336
31.13.6. Vector Single-Width Floating-Point Fused Multiply-Add Instructions ... 336
31.13.7. Vector Widening Floating-Point Fused Multiply-Add Instructions ... 337
31.13.8. Vector Floating-Point Square-Root Instruction ... 338
31.13.9. Vector Floating-Point Reciprocal Square-Root Estimate Instruction ... 338
31.13.10. Vector Floating-Point Reciprocal Estimate Instruction ... 342
31.13.11. Vector Floating-Point MIN/MAX Instructions ... 347
31.13.12. Vector Floating-Point Sign-Injection Instructions ... 347
31.13.13. Vector Floating-Point Compare Instructions ... 348
31.13.14. Vector Floating-Point Classify Instruction ... 349
31.13.15. Vector Floating-Point Merge Instruction ... 349
31.13.16. Vector Floating-Point Move Instruction ... 350
31.13.17. Single-Width Floating-Point/Integer Type-Convert Instructions ... 350
31.13.18. Widening Floating-Point/Integer Type-Convert Instructions ... 350
31.13.19. Narrowing Floating-Point/Integer Type-Convert Instructions ... 351
31.14. Vector Reduction Operations ... 352
31.14.1. Vector Single-Width Integer Reduction Instructions ... 352
31.14.2. Vector Widening Integer Reduction Instructions ... 353
31.14.3. Vector Single-Width Floating-Point Reduction Instructions ... 353
31.14.3.1. Vector Ordered Single-Width Floating-Point Sum Reduction ... 353
31.14.3.2. Vector Unordered Single-Width Floating-Point Sum Reduction ... 354
31.14.3.3. Vector Single-Width Floating-Point Max and Min Reductions ... 354
31.14.4. Vector Widening Floating-Point Reduction Instructions ... 355
31.15. Vector Mask Instructions ... 355
31.15.1. Vector Mask-Register Logical Instructions ... 355
31.15.2. Vector count population in mask `vcpop.m` ... 356
31.15.3. `vfirst` find-first-set mask bit ... 357
31.15.4. `vmsbf.m` set-before-first mask bit ... 357
31.15.5. `vmsif.m` set-including-first mask bit ... 358
31.15.6. `vmsof.m` set-only-first mask bit ... 359
31.15.7. Example using vector mask instructions ... 359
31.15.8. Vector Iota Instruction ... 360
31.15.9. Vector Element Index Instruction ... 362
31.16. Vector Permutation Instructions ... 362
31.16.1. Integer Scalar Move Instructions ... 363
31.16.2. Floating-Point Scalar Move Instructions ... 363
31.16.3. Vector Slide Instructions ... 363
31.16.3.1. Vector Slide-up Instructions ... 364
31.16.3.2. Vector Slide-down Instructions ... 364
31.16.3.3. Vector Slide-1-up ... 365
31.16.3.4. Vector Floating-Point Slide-1-up Instruction ... 365

31.16.3.5. Vector Slide-1-down Instruction 366
31.16.3.6. Vector Floating-Point Slide-1-down Instruction 366
31.16.4. Vector Register Gather Instructions 366
31.16.5. Vector Compress Instruction 367
31.16.5.1. Synthesizing `vdecompress` 368
31.16.6. Whole Vector Register Move 368
31.17. Exception Handling 369
31.17.1. Precise vector traps 369
31.17.2. Imprecise vector traps 370
31.17.3. Selectable precise/imprecise traps 370
31.17.4. Swappable traps 371
31.18. Standard Vector Extensions 371
31.18.1. Zvl*: Minimum Vector Length Standard Extensions 371
31.18.2. Zve*: Vector Extensions for Embedded Processors 371
31.18.3. V: Vector Extension for Application Processors 373
31.18.4. Zvfhmin: Vector Extension for Minimal Half-Precision Floating-Point 373
31.18.5. Zvfh: Vector Extension for Half-Precision Floating-Point 374
31.19. Vector Element Groups 374
31.19.1. Element Group Size 374
31.19.2. Setting `vl` 375
31.19.3. Determining EEW 375
31.19.4. Determining EMUL 375
31.19.5. Element Group Width 376
31.19.6. Masking 376
31.20. Vector Instruction Listing 376
32. Cryptography Extensions: Scalar & Entropy Source Instructions, Version 1.0.1 382
32.1. Introduction 382
32.1.1. Intended Audience 382
32.1.2. Sail Specifications 383
32.1.3. Policies 383
32.2. Extensions Overview 384
32.2.1. `Zbkb` - Bitmanip instructions for Cryptography 384
32.2.2. `Zbkc` - Carry-less multiply instructions 384
32.2.3. `Zbkx` - Crossbar permutation instructions 384
32.2.4. `Zknd` - NIST Suite: AES Decryption 384
32.2.5. `Zkne` - NIST Suite: AES Encryption 385
32.2.6. `Zknh` - NIST Suite: Hash Function Instructions 385
32.2.7. `Zksed` - ShangMi Suite: SM4 Block Cipher Instructions 385
32.2.8. `Zksh` - ShangMi Suite: SM3 Hash Function Instructions 386
32.2.9. `Zkr` - Entropy Source Extension 386
32.2.10. `Zkn` - NIST Algorithm Suite 386
32.2.11. `Zks` - ShangMi Algorithm Suite 386
32.2.12. `Zk` - Standard scalar cryptography extension 387
32.2.13. `Zkt` - Data Independent Execution Latency 387
32.3. Instructions 387

32.3.1. aes32dsi 387
32.3.2. aes32dsmi 389
32.3.3. aes32esi 390
32.3.4. aes32esmi 391
32.3.5. aes64ds 392
32.3.6. aes64dsm 393
32.3.7. aes64es 394
32.3.8. aes64esm 395
32.3.9. aes64im 396
32.3.10. aes64ks1i 397
32.3.11. aes64ks2 398
32.3.12. andn 399
32.3.13. brev8 400
32.3.14. clmul 401
32.3.15. clmulh 402
32.3.16. orn 403
32.3.17. pack 404
32.3.18. packh 405
32.3.19. packw 406
32.3.20. rev8 407
32.3.21. rol 408
32.3.22. rolw 409
32.3.23. ror 410
32.3.24. rori 411
32.3.25. roriw 412
32.3.26. rorw 413
32.3.27. sha256sig0 414
32.3.28. sha256sig1 415
32.3.29. sha256sum0 416
32.3.30. sha256sum1 417
32.3.31. sha512sig0h 418
32.3.32. sha512sig0l 419
32.3.33. sha512sig1h 420
32.3.34. sha512sig1l 421
32.3.35. sha512sum0r 422
32.3.36. sha512sum1r 423
32.3.37. sha512sig0 424
32.3.38. sha512sig1 425
32.3.39. sha512sum0 426
32.3.40. sha512sum1 427
32.3.41. sm3p0 428
32.3.42. sm3p1 429
32.3.43. sm4ed 430
32.3.44. sm4ks 431
32.3.45. unzip 432

32.3.46. xnor 433
32.3.47. xperm8 434
32.3.48. xperm4 435
32.3.49. zip 436
32.4. Entropy Source 437
32.4.1. The `seed` CSR 437
32.4.2. Entropy Source Requirements 438
32.4.2.1. NIST SP 800-90B / FIPS 140-3 Requirements 439
32.4.2.2. BSI AIS-31 PTG.2 / Common Criteria Requirements 439
32.4.2.3. Virtual Sources: Security Requirement 439
32.4.3. Access Control to `seed` 440
32.5. Data Independent Execution Latency Subset: Zkt 440
32.5.1. Scope and Goal 441
32.5.2. Background 441
32.5.3. Specific Instruction Rationale 442
32.5.4. Programming Information 442
32.5.5. Zkt listings 443
32.5.5.1. RVI (Base Instruction Set) 443
32.5.5.2. RVM (Multiply) 444
32.5.5.3. RVC (Compressed) 444
32.5.5.4. Zcb Extension 444
32.5.5.5. RVK (Scalar Cryptography) 445
32.5.5.6. RVB (Bitmanip) 445
32.6. Instruction Rationale 446
32.6.1. AES Instructions 446
32.6.2. SHA2 Instructions 446
32.6.3. SM3 and SM4 Instructions 447
32.6.4. Bitmanip Instructions for Cryptography 447
32.6.4.1. Rotations 447
32.6.4.2. Bit & Byte Permutations 447
32.6.4.3. Carry-less Multiply 448
32.6.4.4. Logic With Negate 448
32.6.4.5. Packing 449
32.6.4.6. Crossbar Permutation Instructions 449
32.7. Entropy Source Rationale and Recommendations 450
32.7.1. Checklists for Design and Self-Certification 450
32.7.2. Standards and Terminology 451
32.7.2.1. Entropy Source (ES) 451
32.7.2.2. Conditioning: Cryptographic and Non-Cryptographic 452
32.7.2.3. Pseudorandom Number Generator (PRNG) 452
32.7.2.4. Deterministic Random Bit Generator (DRBG) 452
32.7.3. Specific Rationale and Considerations 452
32.7.3.1. The `seed` CSR 452
32.7.3.2. NIST SP 800-90B 453
32.7.3.3. BSI AIS-31 453

32.7.3.4. Virtual Sources 454
32.7.3.5. Security Considerations for Direct Hardware Access 454
32.7.4. Security Controls and Health Tests 455
32.7.4.1. T1: On-demand testing 455
32.7.4.2. T2: Continuous checks 455
32.7.4.3. T3: Fatal error states 456
32.7.4.4. Information Flows 456
32.7.5. Implementation Strategies 457
32.7.5.1. Ring Oscillators 457
32.7.5.2. Shot Noise 458
32.7.5.3. Other types of noise 458
32.7.5.4. Continuous Health Tests 458
32.7.5.5. Non-cryptographic Conditioners 459
32.7.5.6. Cryptographic Conditioners 459
32.7.5.7. The Final Random: DRBGs 460
32.7.5.8. Quantum vs. Classical Random 460
32.7.5.9. Post-Quantum Cryptography 461
32.7.6. Suggested GetNoise Test Interface 461
32.8. Supplementary Materials 462
32.9. Supporting Sail Code 462
33. Cryptography Extensions: Vector Instructions, Version 1.0 470
33.1. Introduction 470
33.1.1. Intended Audience 470
33.1.2. Sail Specifications 471
33.1.3. Policies 471
33.1.4. Element Groups 472
33.1.5. Instruction Constraints 473
33.1.6. Vector-Scalar Instructions 474
33.1.7. Software Portability 475
33.2. Extensions Overview 475
33.2.1. `Zvbb` - Vector Basic Bit-manipulation 476
33.2.2. `Zvbc` - Vector Carry-less Multiplication 477
33.2.3. `Zvkb` - Vector Cryptography Bit-manipulation 478
33.2.4. `Zvkg` - Vector GCM/GMAC 479
33.2.5. `Zvkned` - NIST Suite: Vector AES Block Cipher 480
33.2.6. `Zvknh[ab]` - NIST Suite: Vector SHA-2 Secure Hash 481
33.2.7. `Zvksed` - ShangMi Suite: SM4 Block Cipher 482
33.2.8. `Zvksh` - ShangMi Suite: SM3 Secure Hash 483
33.2.9. `Zvkn` - NIST Algorithm Suite 484
33.2.10. `Zvknc` - NIST Algorithm Suite with carry-less multiply 485
33.2.11. `Zvkng` - NIST Algorithm Suite with GCM 486
33.2.12. `Zvks` - ShangMi Algorithm Suite 487
33.2.13. `Zvksc` - ShangMi Algorithm Suite with carry-less multiplication 488
33.2.14. `Zvksg` - ShangMi Algorithm Suite with GCM 489
33.2.15. `Zvkt` - Vector Data-Independent Execution Latency 490

33.2.15.1. All Zvbb instructions 490
33.2.15.2. All Zvbc instructions 490
33.2.15.3. add/sub 490
33.2.15.4. add/sub with carry 491
33.2.15.5. compare and set 491
33.2.15.6. copy 491
33.2.15.7. extend 491
33.2.15.8. logical 491
33.2.15.9. multiply 491
33.2.15.10. multiply-add 492
33.2.15.11. Integer Merge 492
33.2.15.12. permute 492
33.2.15.13. shift 492
33.2.15.14. slide 492
33.3. Instructions 494
33.3.1. vaesdf.[vv,vs] 494
33.3.2. vaesdm.[vv,vs] 496
33.3.3. vaesef.[vv,vs] 498
33.3.4. vaesem.[vv,vs] 500
33.3.5. vaeskf1.vi 502
33.3.6. vaeskf2.vi 504
33.3.7. vaesz.vs 506
33.3.8. vandn.[vv,vx] 508
33.3.9. vbrev.v 510
33.3.10. vbrev8.v 511
33.3.11. vclmul.[vv,vx] 512
33.3.12. vclmulh.[vv,vx] 514
33.3.13. vclz.v 516
33.3.14. vcpop.v 516
33.3.15. vctz.v 517
33.3.16. vghsh.vv 519
33.3.17. vgmul.vv 521
33.3.18. vrev8.v 523
33.3.19. vrol.[vv,vx] 524
33.3.20. vror.[vv,vx,vi] 526
33.3.21. vsha2c[hl].vv 528
33.3.22. vsha2ms.vv 531
33.3.23. vsm3c.vi 534
33.3.24. vsm3me.vv 537
33.3.25. vsm4k.vi 540
33.3.26. vsm4r.[vv,vs] 543
33.3.27. vwsll.[vv,vx,vi] 546
33.4. Crypto Vector Cryptographic Instructions 548
33.5. Vector Bitmanip and Carry-less Multiply Instructions 549
33.6. Supporting Sail Code 552

34. Control-flow Integrity (CFI) **561**
34.1. Landing Pad (Zicfilp) 561
34.1.1. Landing Pad Enforcement 563
34.1.2. Landing Pad Instruction 565
34.2. Shadow Stack (Zicfiss) 565
34.2.1. Zicfiss Instructions Summary 567
34.2.2. Shadow Stack Pointer (`ssp`) 567
34.2.3. Zicfiss Instructions 568
34.2.4. Push to the Shadow Stack 568
34.2.5. Pop from the Shadow Stack 569
34.2.6. Read `ssp` into a Register 572
34.2.7. Atomic Swap from a Shadow Stack Location 574
35. "Zilsd", "Zclsd" Extensions for Load/Store pair for RV32, Version 1.0 ..... **576**
35.1. Load/Store pair instructions (Zilsd) 576
35.2. Compressed Load/Store pair instructions (Zclsd) 576
35.3. Use of x0 as operand 577
35.4. Exception Handling 577
35.5. Instructions 578
35.5.1. ld 578
35.5.2. sd 579
35.5.3. c.ldsp 580
35.5.4. c.sdsp 581
35.5.5. c.ld 582
35.5.6. c.sd 583
36. RV32/64G Instruction Set Listings **584**
37. ISA Extension Naming Conventions **597**
37.1. Case Sensitivity 597
37.2. Base Integer ISA 597
37.3. Instruction-Set Extension Names 597
37.4. Underscores 597
37.5. Additional Standard Unprivileged Extension Names 597
37.6. Supervisor-level Instruction-Set Extension Names 598
37.7. Hypervisor-level Instruction-Set Extension Names 598
37.8. Machine-level Instruction-Set Extension Names 598
37.9. Non-Standard Extension Names 598
37.10. Version Numbers 599
37.11. Subset Naming Convention 599
Appendix A: RVWMO Explanatory Material, Version 0.1 ..... **601**
A.1. Why RVWMO? 601
A.2. Litmus Tests 601
A.3. Explaining the RVWMO Rules 603
A.3.1. Preserved Program Order and Global Memory Order 603
A.3.2. Load value axiom 604
A.3.3. Atomicity axiom 606
A.3.4. Progress axiom 607

A.3.5. Overlapping-Address Orderings (Rules 1-3) 607
A.3.6. Fences (Rule 4) 610
A.3.7. Explicit Synchronization (Rules 5-8) 610
A.3.8. Syntactic Dependencies (Rules 9-11) 612
A.3.9. Pipeline Dependencies (Rules 12-13) 614
A.4. Beyond Main Memory 616
A.4.1. Coherence and Cacheability 616
A.4.2. I/O Ordering 617
A.5. Code Porting and Mapping Guidelines 618
A.6. Implementation Guidelines 623
A.6.1. Possible Future Extensions 625
A.7. Known Issues 626
A.7.1. Mixed-size RSW 626
Appendix B: Formal Memory Model Specifications, Version 0.1 628
B.1. Formal Axiomatic Specification in Alloy 628
B.2. Formal Axiomatic Specification in Herd 632
B.3. An Operational Memory Model 635
B.3.1. Intra-instruction Pseudocode Execution 638
B.3.2. Instruction Instance State 640
B.3.3. Hart State 641
B.3.4. Shared Memory State 641
B.3.5. Transitions 641
B.3.5.1. Fetch instruction 641
B.3.5.2. Initiate memory load operations 642
B.3.5.3. Satisfy memory load operation by forwarding from unpropagated stores 642
B.3.5.4. Satisfy memory load operation from memory 644
B.3.5.5. Complete load operations 644
B.3.5.6. Early `sc` fail 644
B.3.5.7. Paired `sc` 644
B.3.5.8. Initiate memory store operation footprints 644
B.3.5.9. Instantiate memory store operation values 645
B.3.5.10. Commit store instruction 645
B.3.5.11. Propagate store operation 645
B.3.5.12. Commit and propagate store operation of an `sc` 646
B.3.5.13. Late `sc` fail 646
B.3.5.14. Complete store operations 647
B.3.5.15. Satisfy, commit and propagate operations of an AMO 647
B.3.5.16. Commit fence 647
B.3.5.17. Register read 648
B.3.5.18. Register write 648
B.3.5.19. Pseudocode internal step 648
B.3.5.20. Finish instruction 648
B.3.6. Limitations 649
Appendix C: Vector Assembly Code Examples 650
C.1. Vector-vector add example 650

C.2. Example with mixed-width mask and compute. 650
C.3. Memcpy example 651
C.4. Conditional example 651
C.5. SAXPY example 652
C.6. SGEMM example 652
C.7. Division approximation example 657
C.8. Square root approximation example 657
C.9. C standard library strcmp example 658
C.10. Fractional Lmul example 658
Appendix D: Calling Convention for Vector State (Not authoritative - Placeholder Only) 663
Appendix E: Bit Manipulation Extensions Assembly Code Examples 664
E.1. strlen 664
E.2. strcmp 665
Appendix F: Historical Rationale for Extensions 667
F.1. "Zihintpause" Extension for Pause Hint 667
F.2. "Zicond" Extension for Integer Conditional Operations 667
F.3. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions 667
F.4. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0 668
Index 669
Bibliography 671

Preamble

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Arvind, Krste Asanović, Derek Atkins, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Scott Beamer, Abel Bernabeu, Hans Boehm, Alex Bradbury, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, L Peter Deutsch, Ken Dockser, Paul Donahue, Aaron Durbin, Roger Espasa, Greg Favor, Shaked Flur, Stefan Freudenberger, Marc Gauthier, Andy Glew, Jan Gray, Gianluca Guida, Michael Hamburg, John Hauser, Christian Herber, David Horner, Bruce Hoult, Bill Huffman, John Ingalls, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Tariq Kurd, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Yatin Manerkar, Luc Maranget, Ben Marshall, Margaret Martonosi, Phil McCoy, Nathan Menhorn, Christoph Müllner, Joseph Myers, Vijayanand Nagarajan, Torbjørn Viem Ness, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Markku-Juhani O. Saarinen, Albert Ou, John Ousterhout, Daniel Page, David Patterson, Christopher Pulte, Jose Renau, Susmit Sarkar, Josh Scheid, Colin Schmidt, Peter Sewell, Ved Shanbhogue, Brent Spinney, Brendan Sweeney, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Philipp Tomsich, Caroline Trippel, Ray VanDeWalker, Muralidaran Vijayaraghavan, Megan Wachs, Paul Wamsley, Andrew Waterman, Robert Watson, David Weaver, Derek Williams, Claire Wolf, Andrew Wright, Reinoud Zandijk, and Sizhuo Zhang.

This document is released under a Creative Commons Attribution 4.0 International License.

This document is a derivative of "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1" released under the following license: ©2010-2017 Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović. Creative Commons Attribution 4.0 International License. Please cite as: "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019.

Preface

This document describes the RISC-V unprivileged architecture. It contains the following versions of the RISC-V ISA modules, all of which have been ratified:

Base	Version	Status
RV32I	2.1	Ratified
RV32E	2.0	Ratified
RV64E	2.0	Ratified
RV64I	2.1	Ratified
Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
Zicntr	2.0	Ratified
Zihintntl	1.0	Ratified
Zihintpause	2.0	Ratified
Zimop	1.0	Ratified
Zicond	1.0	Ratified
Zilsd	1.0	Ratified
M	2.0	Ratified
Zmmul	1.0	Ratified
A	2.1	Ratified
Zawrs	1.01	Ratified
Zacas	1.0	Ratified
Zabha	1.0	Ratified
Zalasr	1.0	Ratified
RVWMO	2.0	Ratified
Ztso	1.0	Ratified
CMO	1.0	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
Zfh	1.0	Ratified
Zfhmin	1.0	Ratified
BF16	1.0	Ratified
Zfa	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
C	2.0	Ratified

Base	Version	Status
Zce	1.0	Ratified
Zclsd	1.0	Ratified
B	1.0	Ratified
V	1.0	Ratified
Zbkb	1.0	Ratified
Zbkc	1.0	Ratified
Zbkx	1.0	Ratified
Zk	1.0	Ratified
Zks	1.0	Ratified
Zvbb	1.0	Ratified
Zvbc	1.0	Ratified
Zvkg	1.0	Ratified
Zvkned	1.0	Ratified
Zvknhb	1.0	Ratified
Zvksed	1.0	Ratified
Zvksh	1.0	Ratified
Zvkt	1.0	Ratified
Zicfiss	1.0	Ratified
Zicfilp	1.0	Ratified

The changes in this version of the document include:

- Addition of the Zalasr extension for Load-Acquire/Store-Release operations.

Preface to Document Version 20250508

This document describes the RISC-V unprivileged architecture. It contains the following versions of the RISC-V ISA modules, all of which have been ratified:

Base	Version	Status
RV32I	2.1	Ratified
RV32E	2.0	Ratified
RV64E	2.0	Ratified
RV64I	2.1	Ratified
Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
Zicntr	2.0	Ratified
Zihintntl	1.0	Ratified
Zihintpause	2.0	Ratified
Zimop	1.0	Ratified
Zicond	1.0	Ratified

Base	Version	Status
Zilsd	1.0	Ratified
M	2.0	Ratified
Zmmul	1.0	Ratified
A	2.1	Ratified
Zawrs	1.01	Ratified
Zacas	1.0	Ratified
Zabha	1.0	Ratified
RVWMO	2.0	Ratified
Ztso	1.0	Ratified
CMO	1.0	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
Zfh	1.0	Ratified
Zfhmin	1.0	Ratified
BF16	1.0	Ratified
Zfa	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
C	2.0	Ratified
Zce	1.0	Ratified
Zclsd	1.0	Ratified
B	1.0	Ratified
V	1.0	Ratified
Zbkb	1.0	Ratified
Zbkc	1.0	Ratified
Zbkx	1.0	Ratified
Zk	1.0	Ratified
Zks	1.0	Ratified
Zvbb	1.0	Ratified
Zvbc	1.0	Ratified
Zvkg	1.0	Ratified
Zvkned	1.0	Ratified
Zvknhb	1.0	Ratified
Zvksed	1.0	Ratified
Zvksh	1.0	Ratified
Zvkt	1.0	Ratified

Base	Version	Status
Zicfiss	1.0	Ratified
Zicfilp	1.0	Ratified

The changes in this version of the document include:

- The inclusion of all ratified extensions through May 2025.
- Removal of all unratified material.
- Addition of the BFloat16-precision Floating Point extension.
- Addition of the Zabha extension for Byte and Halfword Atomic Memory Operations.

Preface to Document Version 20240411

This document describes the RISC-V unprivileged architecture. It contains the following versions of the RISC-V ISA modules:

Base	Version	Status
RV32I	2.1	Ratified
RV32E	2.0	Ratified
RV64E	2.0	Ratified
RV64I	2.1	Ratified
Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
Zicntr	2.0	Ratified
Zihintntl	1.0	Ratified
Zihintpause	2.0	Ratified
Zimop	1.0	Ratified
Zicond	1.0	Ratified
Zilsd	1.0	Ratified
M	2.0	Ratified
Zmmul	1.0	Ratified
A	2.1	Ratified
Zawrs	1.01	Ratified
Zacas	1.0	Ratified
Zabha	1.0	Ratified
RVWMO	2.0	Ratified
Ztso	1.0	Ratified
CMO	1.0	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
Zfh	1.0	Ratified

Base	Version	Status
Zfhmin	1.0	Ratified
Zfa	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
C	2.0	Ratified
Zce	1.0	Ratified
Zclsd	1.0	Ratified
B	1.0	Ratified
V	1.0	Ratified
Zbkb	1.0	Ratified
Zbkc	1.0	Ratified
Zbkx	1.0	Ratified
Zk	1.0	Ratified
Zks	1.0	Ratified
Zvbb	1.0	Ratified
Zvbc	1.0	Ratified
Zvkg	1.0	Ratified
Zvkned	1.0	Ratified
Zvknhb	1.0	Ratified
Zvksed	1.0	Ratified
Zvksh	1.0	Ratified
Zvkt	1.0	Ratified
Zicfiss	1.0	Ratified
Zicfilp	1.0	Ratified

The changes in this version of the document include:

- The inclusion of all ratified extensions through February 2025.
- The draft Zam extension has been removed, in favor of the definition of a misaligned atomicity granule PMA.
- The concept of vacant memory regions has been superseded by inaccessible memory or I/O regions.
- The removal of unratified content, including the sketch of the RV128I base ISA.

Preface to Document Version 20191213-Base-Ratified

This document describes the RISC-V unprivileged architecture.

The ISA modules marked **Ratified** have been ratified at this time. The modules marked *Frozen* are not expected to change significantly before being put up for ratification. The modules marked *Draft* are expected to change before ratification.

The document contains the following versions of the RISC-V ISA modules:

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	**Version**	**Status**
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

The changes in this version of the document include:

- The A extension, now version 2.1, was ratified by the board in December 2019.
- Defined big-endian ISA variant.
- Moved N extension for user-mode interrupts into Volume II.
- Defined PAUSE hint instruction.

Preface to Document Version 20190608-Base-Ratified

This document describes the RISC-V unprivileged architecture.

The RVWMO memory model has been ratified at this time. The ISA modules marked **Ratified**, have been ratified at this time. The modules marked *Frozen* are not expected to change significantly before being put up for ratification. The modules marked *Draft* are expected to change before ratification.

The document contains the following versions of the RISC-V ISA modules:

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	**Version**	**Status**
Zifencei	**2.0**	**Ratified**
Zicsr	**2.0**	**Ratified**
M	**2.0**	**Ratified**
A	*2.0*	Frozen
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zam	*0.1*	*Draft*

The changes in this version of the document include:

- Moved description to **Ratified** for the ISA modules ratified by the board in early 2019.
- Removed the A extension from ratification.
- Changed document version scheme to avoid confusion with versions of the ISA modules.
- Incremented the version numbers of the base integer ISA to 2.1, reflecting the presence of the ratified RVWMO memory model and exclusion of FENCE.I, counters, and CSR instructions that were in previous base ISA.
- Incremented the version numbers of the F and D extensions to 2.2, reflecting that version 2.1 changed the canonical NaN, and version 2.2 defined the NaN-boxing scheme and changed the definition of the FMIN and FMAX instructions.
- Changed name of document to refer to "unprivileged" instructions as part of move to separate ISA specifications from platform profile mandates.
- Added clearer and more precise definitions of execution environments, harts, traps, and memory accesses.
- Defined instruction-set categories: *standard*, *reserved*, *custom*, *non-standard*, and *non-conforming*.
- Removed text implying operation under alternate endianness, as alternate-endianness operation has not yet been defined for RISC-V.

- Changed description of misaligned load and store behavior. The specification now allows visible misaligned address traps in execution environment interfaces, rather than just mandating invisible handling of misaligned loads and stores in user mode. Also, now allows access-fault exceptions to be reported for misaligned accesses (including atomics) that should not be emulated.
- Moved FENCE.I out of the mandatory base and into a separate extension, with Zifencei ISA name. FENCE.I was removed from the Linux user ABI and is problematic in implementations with large incoherent instruction and data caches. However, it remains the only standard instruction-fetch coherence mechanism.
- Removed prohibitions on using RV32E with other extensions.
- Removed platform-specific mandates that certain encodings produce illegal-instruction exceptions in RV32E and RV64I chapters.
- Counter/timer instructions are now not considered part of the mandatory base ISA, and so CSR instructions were moved into separate chapter and marked as version 2.0, with the unprivileged counters moved into another separate chapter. The counters are not ready for ratification as there are outstanding issues, including counter inaccuracies.
- A CSR-access ordering model has been added.
- Explicitly defined the 16-bit half-precision floating-point format for floating-point instructions in the 2-bit *fmt field*.
- Defined the signed-zero behavior of FMIN.*fmt* and FMAX.*fmt*, and changed their behavior on signaling-NaN inputs to conform to the `minimumNumber` and `maximumNumber` operations in the proposed IEEE 754-201x specification.
- The memory consistency model, RVWMO, has been defined.
- The "Zam" extension, which permits misaligned AMOs and specifies their semantics, has been defined.
- The "Ztso" extension, which enforces a stricter memory consistency model than RVWMO, has been defined.
- Improvements to the description and commentary.
- Defined the term `IALIGN` as shorthand to describe the instruction-address alignment constraint.
- Removed text of `P` extension chapter as now superseded by active task group documents.
- Removed text of `V` extension chapter as now superseded by separate vector extension draft document.

Preface to Document Version 2.2

This is version 2.2 of the document describing the RISC-V user-level architecture. The document contains the following versions of the RISC-V ISA modules:

Base	*Version*	*Draft Frozen?*
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	**Version**	**Frozen?**
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y

Base	*Version*	*Draft Frozen?*
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

To date, no parts of the standard have been officially ratified by the RISC-V Foundation, but the components labeled "frozen" above are not expected to change during the ratification process beyond resolving ambiguities and holes in the specification.

The major changes in this version of the document include:

- The previous version of this document was released under a Creative Commons Attribution 4.0 International License by the original authors, and this and future versions of this document will be released under the same license.
- Rearranged chapters to put all extensions first in canonical order.
- Improvements to the description and commentary.
- Modified implicit hinting suggestion on `JALR` to support more efficient macro-op fusion of `LUI/JALR` and `AUIPC/JALR` pairs.
- Clarification of constraints on load-reserved/store-conditional sequences.
- A new table of control and status register (CSR) mappings.
- Clarified purpose and behavior of high-order bits of `fcsr`.
- Corrected the description of the `FNMADD`.*fmt* and `FNMSUB`.*fmt* instructions, which had suggested the incorrect sign of a zero result.
- Instructions `FMV.S.X` and `FMV.X.S` were renamed to `FMV.W.X` and `FMV.X.W` respectively to be more consistent with their semantics, which did not change. The old names will continue to be supported in the tools.
- Specified behavior of narrower (<FLEN) floating-point values held in wider `f` registers using NaN-boxing model.
- Defined the exception behavior of FMA(∞, 0, qNaN).
- Added note indicating that the `P` extension might be reworked into an integer packed-SIMD proposal for fixed-point operations using the integer registers.
- A draft proposal of the V vector instruction-set extension.
- An early draft proposal of the N user-level traps extension.
- An expanded pseudoinstruction listing.
- Removal of the calling convention chapter, which has been superseded by the RISC-V ELF psABI Specification (*RISC-V ELF PsABI Specification*, n.d.).
- The C extension has been frozen and renumbered version 2.0.

Preface to Document Version 2.1

This is version 2.1 of the document describing the RISC-V user-level architecture. Note the frozen user-level ISA base and extensions `IMAFDQ` version 2.0 have not changed from the previous version of this document (Waterman et al., 2014), but some specification holes have been fixed and the documentation has been improved. Some changes have been made to the software conventions.

- Numerous additions and improvements to the commentary sections.
- Separate version numbers for each chapter.
- Modification to long instruction encodings >64 bits to avoid moving the *rd* specifier in very long instruction formats.
- CSR instructions are now described in the base integer format where the counter registers are introduced, as opposed to only being introduced later in the floating-point section (and the companion privileged architecture manual).
- The SCALL and SBREAK instructions have been renamed to `ECALL` and `EBREAK`, respectively. Their encoding and functionality are unchanged.
- Clarification of floating-point NaN handling, and a new canonical NaN value.
- Clarification of values returned by floating-point to integer conversions that overflow.
- Clarification of `LR/SC` allowed successes and required failures, including use of compressed instructions in the sequence.
- A new `RV32E` base ISA proposal for reduced integer register counts, supports `MAC` extensions.
- A revised calling convention.
- Relaxed stack alignment for soft-float calling convention, and description of the RV32E calling convention.
- A revised proposal for the `C` compressed extension, version 1.9 .

Preface to Version 2.0

This is the second release of the user ISA specification, and we intend the specification of the base user ISA plus general extensions (i.e., IMAFD) to remain fixed for future development. The following changes have been made since Version 1.0 (Waterman et al., 2011) of this ISA specification.

- The ISA has been divided into an integer base with several standard extensions.
- The instruction formats have been rearranged to make immediate encoding more efficient.
- The base ISA has been defined to have a little-endian memory system, with big-endian or bi-endian as non-standard variants.
- Load-Reserved/Store-Conditional (`LR/SC`) instructions have been added in the atomic instruction extension.
- `AMOs` and `LR/SC` can support the release consistency model.
- The `FENCE` instruction provides finer-grain memory and I/O orderings.
- An `AMO` for fetch-and-`XOR` (`AMOXOR`) has been added, and the encoding for `AMOSWAP` has been changed to make room.
- The `AUIPC` instruction, which adds a 20-bit upper immediate to the `PC`, replaces the `RDNPC` instruction, which only read the current `PC` value. This results in significant savings for position-independent code.
- The `JAL` instruction has now moved to the `U-Type` format with an explicit destination register, and the `J` instruction has been dropped being replaced by `JAL` with *rd*=`x0`. This removes the only instruction with

an implicit destination register and removes the `J-Type` instruction format from the base ISA. There is an accompanying reduction in `JAL` reach, but a significant reduction in base ISA complexity.

- The static hints on the `JALR` instruction have been dropped. The hints are redundant with the *rd* and *rs1* register specifiers for code compliant with the standard calling convention.
- The `JALR` instruction now clears the lowest bit of the calculated target address, to simplify hardware and to allow auxiliary information to be stored in function pointers.
- The `MFTX.S` and `MFTX.D` instructions have been renamed to `FMV.X.S` and `FMV.X.D`, respectively. Similarly, `MXTF.S` and `MXTF.D` instructions have been renamed to `FMV.S.X` and `FMV.D.X`, respectively.
- The `MFFSR` and `MTFSR` instructions have been renamed to `FRCSR` and `FSCSR`, respectively. `FRRM`, `FSRM`, `FRFLAGS`, and `FSFLAGS` instructions have been added to individually access the rounding mode and exception flags subfields of the `fcsr`.
- The `FMV.X.S` and `FMV.X.D` instructions now source their operands from *rs1*, instead of *rs2*. This change simplifies datapath design.
- `FCLASS.S` and `FCLASS.D` floating-point classify instructions have been added.
- A simpler NaN generation and propagation scheme has been adopted.
- For `RV32I`, the system performance counters have been extended to 64-bits wide, with separate read access to the upper and lower 32 bits.
- Canonical `NOP` and `MV` encodings have been defined.
- Standard instruction-length encodings have been defined for 48-bit, 64-bit, and >64-bit instructions.
- Description of a 128-bit address space variant, `RV128`, has been added.
- Major opcodes in the 32-bit base instruction format have been allocated for user-defined custom extensions.
- A typographical error that suggested that stores source their data from *rd* has been corrected to refer to *rs2*.

Chapter 1. Introduction

RISC-V (pronounced "risk-five") is a new instruction-set architecture (ISA) that was originally designed to support computer architecture research and education, but which we now hope will also become a standard free and open architecture for industry implementations. Our goals in defining RISC-V include:

- A completely *open* ISA that is freely available to academia and industry.
- A *real* ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- An ISA separated into a *small* base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development.
- Support for the revised 2008 IEEE-754 floating-point standard. (*ANSI/IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic*, 2008)
- An ISA supporting extensive ISA extensions and specialized variants.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- An ISA with support for highly parallel multicore or manycore implementations, including heterogeneous multiprocessors.
- Optional *variable-length instructions* to both expand available instruction encoding space and to support an optional *dense instruction encoding* for improved performance, static code size, and energy efficiency.
- A fully virtualizable ISA to ease hypervisor development.
- An ISA that simplifies experiments with new privileged architecture designs.

Commentary on our design decisions is formatted as in this paragraph. This non-normative text can be skipped if the reader is only interested in the specification itself.

The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I (Patterson & Séquin, 1981), RISC-II (Katevenis et al., 1983), SOAR (Ungar et al., 1984), and SPUR (Lee et al., 1989) were the first four). We also pun on the use of the Roman numeral "V" to signify "variations" and "vectors", as support for a range of architecture research, including various data-parallel accelerators, is an explicit goal of the ISA design.

The RISC-V ISA is defined avoiding implementation details as much as possible (although commentary is included on implementation-driven decisions) and should be read as the software-visible interface to a wide variety of implementations rather than as the design of a particular hardware artifact. The RISC-V manual is structured in two volumes. This volume covers the design of the base *unprivileged* instructions, including optional unprivileged ISA extensions. Unprivileged instructions are those that are generally usable in all privilege modes in all privileged architectures, though behavior might vary depending on privilege mode and privilege architecture. The second volume provides the design of the first ("classic") privileged architecture. The manuals use IEC 80000-13:2008 conventions, with a byte of 8 bits.

In the unprivileged ISA design, we tried to remove any dependence on particular microarchitectural features, such as cache line size, or on privileged architecture details, such as page translation. This is both for simplicity and to allow maximum flexibility for alternative microarchitectures or alternative privileged architectures.

1.1. RISC-V Hardware Platform Terminology

A RISC-V hardware platform can contain one or more RISC-V-compatible processing cores together with other non-RISC-V-compatible cores, fixed-function accelerators, various physical memory structures, I/O devices, and an interconnect structure to allow the components to communicate.

A component is termed a *core* if it contains an independent instruction fetch unit. A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading.

A RISC-V core might have additional specialized instruction-set extensions or an added *coprocessor*. We use the term *coprocessor* to refer to a unit that is attached to a RISC-V core and is mostly sequenced by a RISC-V instruction stream, but which contains additional architectural state and instruction-set extensions, and possibly some limited autonomy relative to the primary RISC-V instruction stream.

We use the term *accelerator* to refer to either a non-programmable fixed-function unit or a core that can operate autonomously but is specialized for certain tasks. In RISC-V systems, we expect many programmable accelerators will be RISC-V-based cores with specialized instruction-set extensions and/or customized coprocessors. An important class of RISC-V accelerators are I/O accelerators, which offload I/O processing tasks from the main application cores.

The system-level organization of a RISC-V hardware platform can range from a single-core microcontroller to a many-thousand-node cluster of shared-memory manycore server nodes. Even small systems-on-a-chip might be structured as a hierarchy of multicomputers and/or multiprocessors to modularize development effort or to provide secure isolation between subsystems.

1.2. RISC-V Software Execution Environments and Harts

The behavior of a RISC-V program depends on the execution environment in which it runs. A RISC-V execution environment interface (EEI) defines the initial state of the program, the number and type of harts in the environment including the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart (i.e., the ISA is one component of the EEI), and the handling of any interrupts or exceptions raised during execution including environment calls. Examples of EEIs include the Linux application binary interface (ABI), or the RISC-V supervisor binary interface (SBI). The implementation of a RISC-V execution environment can be pure hardware, pure software, or a combination of hardware and software. For example, opcode traps and software emulation can be used to implement functionality not provided in hardware. Examples of execution environment implementations include:

- "Bare metal" hardware platforms where harts are directly implemented by physical processor threads and instructions have full access to the physical address space. The hardware platform defines an execution environment that begins at power-on reset.
- RISC-V operating systems that provide multiple user-level execution environments by multiplexing user-level harts onto available physical processor threads and by controlling access to memory via virtual memory.
- RISC-V hypervisors that provide multiple supervisor-level execution environments for guest operating systems.
- RISC-V emulators, such as Spike, QEMU or rv8, which emulate RISC-V harts on an underlying x86 system, and which can provide either a user-level or a supervisor-level execution environment.

A bare hardware platform can be considered to define an EEI, where the accessible harts, memory, and other devices populate the environment, and the initial state is that at power-on reset. Generally, most software is designed to use a more abstract interface to the hardware, as more abstract EEIs provide greater portability across different hardware platforms. Often EEIs

are layered on top of one another, where one higher-level EEI uses another lower-level EEI.

From the perspective of software running in a given execution environment, a hart is a resource that autonomously fetches and executes RISC-V instructions within that execution environment. In this respect, a hart behaves like a hardware thread resource even if time-multiplexed onto real hardware by the execution environment. Some EEIs support the creation and destruction of additional harts, for example, via environment calls to fork new harts.

The execution environment is responsible for ensuring the eventual forward progress of each of its harts. For a given hart, that responsibility is suspended while the hart is exercising a mechanism that explicitly waits for an event, such as the wait-for-interrupt instruction defined in Volume II of this specification; and that responsibility ends if the hart is terminated. The following events constitute forward progress:

- The retirement of an instruction.
- A trap, as defined in Section 1.6.
- Any other event defined by an extension to constitute forward progress.

The term hart was introduced in the work on Lithe (Pan et al., 2009) and (Pan et al., 2010) to provide a term to represent an abstract execution resource as opposed to a software thread programming abstraction.

The important distinction between a hardware thread (hart) and a software thread context is that the software running inside an execution environment is not responsible for causing progress of each of its harts; that is the responsibility of the outer execution environment. So the environment's harts operate like hardware threads from the perspective of the software inside the execution environment.

An execution environment implementation might time-multiplex a set of guest harts onto fewer host harts provided by its own execution environment but must do so in a way that guest harts operate like independent hardware threads. In particular, if there are more guest harts than host harts then the execution environment must be able to preempt the guest harts and must not wait indefinitely for guest software on a guest hart to "yield" control of the guest hart.

1.3. RISC-V ISA Overview

A RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Although it is convenient to speak of *the* RISC-V ISA, RISC-V is actually a family of related ISAs, of which there are currently four base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, described in Chapter 2 and Chapter 4, which provide 32-bit or 64-bit address spaces respectively. We use the term XLEN to refer to the width of an integer register in bits (either 32 or 64). Chapter 3 describes the RV32E and RV64E subset variants of the RV32I or RV64I base instruction sets respectively, which have been added to support small microcontrollers, and which have half the number of integer registers. The base integer instruction sets use a two's-complement representation for signed integer values.

Although 64-bit address spaces are a requirement for larger systems, we believe 32-bit address spaces will remain adequate for many embedded and client devices for decades to come and will be desirable to lower memory traffic and energy consumption. In addition, 32-bit address spaces are sufficient for educational purposes. A larger flat 128-bit address space might eventually be required and could be accommodated with a new RV128I base ISA within the existing RISC-V ISA framework.

The four base ISAs in RISC-V are treated as distinct base ISAs. A common question is why is there not a single ISA, and in particular, why is RV32I not a strict subset of RV64I? Some earlier ISA designs (SPARC, MIPS) adopted a strict superset policy when increasing address space size to support running existing 32-bit binaries on new 64-bit hardware.

The main advantage of explicitly separating base ISAs is that each base ISA can be optimized for its needs without requiring to support all the operations needed for other base ISAs. For example, RV64I can omit instructions and CSRs that are only needed to cope with the narrower registers in RV32I. The RV32I variants can use encoding space otherwise reserved for instructions only required by wider address-space variants.

The main disadvantage of not treating the design as a single ISA is that it complicates the hardware needed to emulate one base ISA on another (e.g., RV32I on RV64I). However, differences in addressing and illegal-instruction traps generally mean some mode switch would be required in hardware in any case even with full superset instruction encodings, and the different RISC-V base ISAs are similar enough that supporting multiple versions is relatively low cost. Although some have proposed that the strict superset design would allow legacy 32-bit libraries to be linked with 64-bit code, this is impractical in practice, even with compatible encodings, due to the differences in software calling conventions and system-call interfaces.

The RISC-V privileged architecture provides fields in `misa` *to control the unprivileged ISA at each level to support emulating different base ISAs on the same hardware. We note that newer SPARC and MIPS ISA revisions have deprecated support for running 32-bit code unchanged on 64-bit systems.*

*A related question is why there is a different encoding for 32-bit adds in RV32I (ADD) and RV64I (ADDW)? The ADDW opcode could be used for 32-bit adds in RV32I and ADDD for 64-bit adds in RV64I, instead of the existing design which uses the same opcode ADD for 32-bit adds in RV32I and 64-bit adds in RV64I with a different opcode ADDW for 32-bit adds in RV64I. This would also be more consistent with the use of the same LW opcode for 32-bit load in both RV32I and RV64I. The very first versions of RISC-V ISA did have a variant of this alternate design, but the RISC-V design was changed to the current choice in January 2011. Our focus was on supporting 32-bit integers in the 64-bit ISA, not on providing compatibility with the 32-bit ISA, and the motivation was to remove the asymmetry that arose from having not all opcodes in RV32I have a *W suffix (e.g., ADDW, but AND not ANDW). In hindsight, this was perhaps not well-justified and a consequence of designing both ISAs at the same time as opposed to adding one later to sit on top of another, and also from a belief we had to fold platform requirements into the ISA spec which would imply that all the RV32I instructions would have been required in RV64I. It is too late to change the encoding now, but this is also of little practical consequence for the reasons stated above.*

*It has been noted we could enable the *W variants as an extension to RV32I systems to provide a common encoding across RV64I and a future RV32 variant.*

RISC-V has been designed to support extensive customization and specialization. Each base integer ISA can be extended with one or more optional instruction-set extensions. An extension may be categorized as either standard, custom, or non-conforming. For this purpose, we divide each RISC-V instruction-set

encoding space (and related encoding spaces such as the CSRs) into three disjoint categories: *standard*, *reserved*, and *custom*. Standard extensions and encodings are defined by RISC-V International; any extensions not defined by RISC-V International are *non-standard*. Each base ISA and its standard extensions use only standard encodings, and shall not conflict with each other in their uses of these encodings. Reserved encodings are currently not defined but are saved for future standard extensions; once thus used, they become standard encodings. Custom encodings shall never be used for standard extensions and are made available for vendor-specific non-standard extensions. Non-standard extensions are either custom extensions, that use only custom encodings, or *non-conforming* extensions, that use any standard or reserved encoding. Instruction-set extensions are generally shared but may provide slightly different functionality depending on the base ISA. We have also developed a naming convention for RISC-V base instructions and instruction-set extensions, described in detail in Chapter 37.

To support more general software development, a set of standard extensions are defined to provide integer multiply/divide, atomic operations, and single and double-precision floating-point arithmetic. The base integer ISA is named "I" (prefixed by RV32 or RV64 depending on integer register width), and contains integer computational instructions, integer loads, integer stores, and control-flow instructions. The standard integer multiplication and division extension is named "M", and adds instructions to multiply and divide values held in the integer registers. The standard atomic instruction extension, denoted by "A", adds instructions that atomically read, modify, and write memory for inter-processor synchronization. The standard single-precision floating-point extension, denoted by "F", adds floating-point registers, single-precision computational instructions, and single-precision loads and stores. The standard double-precision floating-point extension, denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores. The standard "C" compressed instruction extension provides narrower 16-bit forms of common instructions.

Beyond the base integer ISA and these standard extensions, we believe it is rare that a new instruction will provide a significant benefit for all applications, although it may be very beneficial for a certain domain. As energy efficiency concerns are forcing greater specialization, we believe it is important to simplify the required portion of an ISA specification. Whereas other architectures usually treat their ISA as a single entity, which changes to a new version as instructions are added over time, RISC-V will endeavor to keep the base and each standard extension constant over time, and instead layer new instructions as further optional extensions. For example, the base integer ISAs will continue as fully supported standalone ISAs, regardless of any subsequent extensions.

1.4. Memory

A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes for all memory accesses. A *word* of memory is defined as 32 bits (4 bytes). Correspondingly, a *halfword* is 16 bits (2 bytes), a *doubleword* is 64 bits (8 bytes), and a *quadword* is 128 bits (16 bytes). The memory address space is circular, so that the byte at address $2^{XLEN}-1$ is adjacent to the byte at address zero. Accordingly, memory address computations done by the hardware ignore overflow and instead wrap around modulo 2^{XLEN}.

The execution environment determines the mapping of hardware resources into a hart's address space. Different address ranges of a hart's address space may (1) contain *main memory*, or (2) contain one or more *I/O devices*. Reads and writes of I/O devices may have visible side effects, but accesses to main memory cannot. Vacant address ranges are not a separate category but can be represented as either main memory or I/O regions that are not accessible. Although it is possible for the execution environment to call everything in a hart's address space an I/O device, it is usually expected that some portion will be specified as main memory.

When a RISC-V platform has multiple harts, the address spaces of any two harts may be entirely the same, or entirely different, or may be partly different but sharing some subset of resources, mapped into the same or different address ranges.

For a purely "bare metal" environment, all harts may see an identical address space, accessed entirely by physical addresses. However, when the execution environment includes an operating system employing address translation, it is common for each hart to be given a virtual address space that is largely or entirely its own.

Executing each RISC-V machine instruction entails one or more memory accesses, subdivided into *implicit* and *explicit* accesses. For each instruction executed, an *implicit* memory read (instruction fetch) is done to obtain the encoded instruction to execute. Many RISC-V instructions perform no further memory accesses beyond instruction fetch. Specific load and store instructions perform an *explicit* read or write of memory at an address determined by the instruction. The execution environment may dictate that instruction execution performs other *implicit* memory accesses (such as to implement address translation) beyond those documented for the unprivileged ISA.

The execution environment determines what portions of the address space are accessible for each kind of memory access. For example, the set of locations that can be implicitly read for instruction fetch may or may not have any overlap with the set of locations that can be explicitly read by a load instruction; and the set of locations that can be explicitly written by a store instruction may be only a subset of locations that can be read. Ordinarily, if an instruction attempts to access memory at an inaccessible address, an exception is raised for the instruction.

Except when specified otherwise, implicit reads that do not raise an exception and that have no side effects may occur arbitrarily early and speculatively, even before the machine could possibly prove that the read will be needed. For instance, a valid implementation could attempt to read all of main memory at the earliest opportunity, cache as many fetchable (executable) bytes as possible for later instruction fetches, and avoid reading main memory for instruction fetches ever again. To ensure that certain implicit reads are ordered only after writes to the same memory locations, software must execute specific fence or cache-control instructions defined for this purpose (such as the FENCE.I instruction defined in Chapter 5).

The memory accesses (implicit or explicit) made by a hart may appear to occur in a different order as perceived by another hart or by any other agent that can access the same memory. This perceived reordering of memory accesses is always constrained, however, by the applicable memory consistency model. The default memory consistency model for RISC-V is the RISC-V Weak Memory Ordering (RVWMO), defined in Chapter 18 and in appendices. Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in Chapter 19. The execution environment may also add constraints that further limit the perceived reordering of memory accesses. Since the RVWMO model is the weakest model allowed for any RISC-V implementation, software written for this model is compatible with the actual memory consistency rules of all RISC-V implementations. As with implicit reads, software must execute fence or cache-control instructions to ensure specific ordering of memory accesses beyond the requirements of the assumed memory consistency model and execution environment.

1.5. Base Instruction-Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction *parcels* in length and parcels are naturally aligned on 16-bit boundaries. The standard compressed ISA extension described in Chapter 28 reduces code size by providing compressed 16-bit instructions and relaxes the alignment constraints to allow all instructions (16 bit and 32 bit) to be aligned on any 16-bit boundary to improve code density.

We use the term IALIGN (measured in bits) to refer to the instruction-address alignment constraint the implementation enforces. IALIGN is 32 bits in the base ISA, but some ISA extensions, including the

compressed ISA extension, relax IALIGN to 16 bits. IALIGN may not take on any value other than 16 or 32.

We use the term ILEN (measured in bits) to refer to the maximum instruction length supported by an implementation, and which is always a multiple of IALIGN. For implementations supporting only a base instruction set, ILEN is 32 bits. Implementations supporting longer instructions have larger values of ILEN.

All the 32-bit instructions in the base ISA have their lowest two bits set to `11`. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to `00`, `01`, or `10`.

Given the code size and energy savings of a compressed format, we wanted to build in support for a compressed format to the ISA encoding scheme rather than adding this as an afterthought, but to allow simpler implementations we didn't want to make the compressed format mandatory. We also wanted to optionally allow longer instructions to support experimentation and larger instruction-set extensions. Although our encoding convention required a tighter encoding of the core RISC-V ISA, this has several beneficial effects.

An implementation of the standard IMAFD ISA need only hold the most-significant 30 bits in instruction caches (a 6.25% saving). On instruction cache refills, any instructions encountered with either low bit clear should be recoded into illegal 30-bit instructions before storing in the cache to preserve illegal-instruction exception behavior.

Perhaps more importantly, by condensing our base ISA into a subset of the 32-bit instruction word, we leave more space available for non-standard and custom extensions. In particular, the base RV32I ISA uses less than 1/8 of the encoding space in the 32-bit instruction word. An implementation that does not require support for the standard compressed instruction extension can map 3 additional non-conforming 30-bit instruction spaces into the 32-bit fixed-width format, while preserving support for standard ≥32-bit instruction-set extensions.

Encodings with bits [15:0] all zeros are defined as illegal instructions. These instructions are considered to be of minimal length: 16 bits if any 16-bit instruction-set extension is present, otherwise 32 bits. The encoding with bits [ILEN-1:0] all ones is also illegal; this instruction is considered to be ILEN bits long.

We consider it a feature that any length of instruction containing all zero bits is not legal, as this quickly traps erroneous jumps into zeroed memory regions. Similarly, we also reserve the instruction encoding containing all ones to be an illegal instruction, to catch the other common pattern observed with unprogrammed non-volatile memory devices, disconnected memory buses, or broken memory devices.

Software can rely on a naturally aligned 32-bit word containing zero to act as an illegal instruction on all RISC-V implementations, to be used by software where an illegal instruction is explicitly desired. Defining a corresponding known illegal value for all ones is more difficult due to the variable-length encoding. Software cannot generally use the illegal value of ILEN bits of all 1s, as software might not know ILEN for the eventual target machine (e.g., if software is compiled into a standard binary library used by many different machines). Defining a 32-bit word of all ones as illegal was also considered, as all machines must support a 32-bit instruction size, but this requires the instruction-fetch unit on machines with ILEN >32 report an illegal-instruction exception rather than an access-fault exception when such an instruction borders a protection boundary, complicating variable-instruction-length fetch and decode.

RISC-V base ISAs have either little-endian or big-endian memory systems, with the privileged architecture further defining bi-endian operation. Instructions are stored in memory as a sequence of 16-bit little-endian parcels, regardless of memory system endianness. Parcels forming one instruction are stored at increasing halfword addresses, with the lowest-addressed parcel holding the lowest-numbered bits in the instruction specification.

We originally chose little-endian byte ordering for the RISC-V memory system because little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). A minor point is that we have also found little-endian memory systems to be more natural for hardware designers. However, certain application areas, such as IP networking, operate on big-endian data structures, and certain legacy code bases have been built assuming big-endian processors, so we have defined big-endian and bi-endian variants of RISC-V.

We have to fix the order in which instruction parcels are stored in memory, independent of memory system endianness, to ensure that the length-encoding bits always appear first in halfword address order. This allows the length of a variable-length instruction to be quickly determined by an instruction-fetch unit by examining only the first few bits of the first 16-bit instruction parcel.

We further make the instruction parcels themselves little-endian to decouple the instruction encoding from the memory system endianness altogether. This design benefits both software tooling and bi-endian hardware. Otherwise, for instance, a RISC-V assembler or disassembler would always need to know the intended active endianness, despite that in bi-endian systems, the endianness mode might change dynamically during execution. In contrast, by giving instructions a fixed endianness, it is sometimes possible for carefully written software to be endianness-agnostic even in binary form, much like position-independent code.

The choice to have instructions be only little-endian does have consequences, however, for RISC-V software that encodes or decodes machine instructions. Big-endian JIT compilers, for example, must swap the byte order when storing to instruction memory.

Once we had decided to fix on a little-endian instruction encoding, this naturally led to placing the length-encoding bits in the LSB positions of the instruction format to avoid breaking up opcode fields.

1.6. Exceptions, Traps, and Interrupts

We use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart. We use the term *interrupt* to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control. We use the term *trap* to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.

The instruction descriptions in following chapters describe conditions that can raise an exception during execution. The general behavior of most RISC-V EEIs is that a trap to some handler occurs when an exception is signaled on an instruction (except for floating-point exceptions, which, in the standard floating-point extensions, do not cause traps). The manner in which interrupts are generated, routed to, and enabled by a hart depends on the EEI.

Our use of "exception" and "trap" is compatible with that in the IEEE-754 floating-point standard.

How traps are handled and made visible to software running on the hart depends on the enclosing execution environment. From the perspective of software running inside an execution environment, traps encountered by a hart at runtime can have four different effects:

Contained Trap

The trap is visible to, and handled by, software running inside the execution environment. For example, in an EEI providing both supervisor and user mode on harts, an ECALL by a user-mode hart will generally result in a transfer of control to a supervisor-mode handler running on the same hart.

Similarly, in the same environment, when a hart is interrupted, an interrupt handler will be run in supervisor mode on the hart.

Requested Trap

The trap is a synchronous exception that is an explicit call to the execution environment requesting an action on behalf of software inside the execution environment. An example is a system call. In this case, execution may or may not resume on the hart after the requested action is taken by the execution environment. For example, a system call could remove the hart or cause an orderly termination of the entire execution environment.

Invisible Trap

The trap is handled transparently by the execution environment and execution resumes normally after the trap is handled. Examples include emulating missing instructions, handling non-resident page faults in a demand-paged virtual-memory system, or handling device interrupts for a different job in a multiprogrammed machine. In these cases, the software running inside the execution environment is not aware of the trap (we ignore timing effects in these definitions).

Fatal Trap

The trap represents a fatal failure and causes the execution environment to terminate execution. Examples include failing a virtual-memory page-protection check or allowing a watchdog timer to expire. Each EEI should define how execution is terminated and reported to an external environment.

Table 1 shows the characteristics of each kind of trap.

Table 1. Characteristics of traps

	Contained	Requested	Invisible	Fatal
Execution terminates	No	No[1]	No	Yes
Software is oblivious	No	No	Yes	Yes[2]
Handled by environment	No	Yes	Yes	Yes

[1] Termination may be requested
[2] Imprecise fatal traps might be observable by software

The EEI defines for each trap whether it is handled precisely, though the recommendation is to maintain preciseness where possible. Contained and requested traps can be observed to be imprecise by software inside the execution environment. Invisible traps, by definition, cannot be observed to be precise or imprecise by software running inside the execution environment. Fatal traps can be observed to be imprecise by software running inside the execution environment, if known-errorful instructions do not cause immediate termination.

Because this document describes unprivileged instructions, traps are rarely mentioned. Architectural means to handle contained traps are defined in the privileged architecture manual, along with other features to support richer EEIs. Unprivileged instructions that are defined solely to cause requested traps are documented here. Invisible traps are, by their nature, out of scope for this document. Instruction encodings that are not defined here and not defined by some other means may cause a fatal trap.

1.7. UNSPECIFIED Behaviors and Values

The architecture fully describes what implementations must do and any constraints on what they may do. In cases where the architecture intentionally does not constrain implementations, the term UNSPECIFIED is explicitly used.

The term UNSPECIFIED refers to a behavior or value that is intentionally unconstrained. The definition of these behaviors or values is open to extensions, platform standards, or implementations. Extensions, platform standards, or implementation documentation may provide normative content to further constrain cases that the base architecture defines as UNSPECIFIED.

Like the base architecture, extensions should fully describe allowable behavior and values and use the term UNSPECIFIED for cases that are intentionally unconstrained. These cases may be constrained or defined by other extensions, platform standards, or implementations.

Chapter 2. RV32I Base Integer Instruction Set, Version 2.1

This chapter describes the RV32I base integer instruction set.

RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might cover the ECALL/EBREAK instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE instruction as a NOP, reducing base instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).

In practice, a hardware implementation including the machine-mode privileged architecture will also require the 6 CSR instructions.

Subsets of the base integer ISA might be useful for pedagogical purposes, but the base has been defined such that there should be little incentive to subset a real hardware implementation beyond omitting support for misaligned memory accesses and treating all SYSTEM instructions as a single trap.

The standard RISC-V assembly language syntax is documented in the Assembly Programmer's Manual (RISC-V Assembly Programmer's Manual, n.d.).

Most of the commentary for RV32I also applies to the RV64I base.

2.1. Programmers' Model for Base Integer ISA

Table 2 shows the unprivileged state for the base integer ISA. For RV32I, the 32 `x` registers are each 32 bits wide, i.e., `XLEN=32`. Register `x0` is hardwired with all bits equal to 0. General purpose registers `x1-x31` hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers.

There is one additional unprivileged register: the program counter `pc` holds the address of the current instruction.

Table 2. RISC-V base unprivileged integer register state.

XLEN-1	0
x0/zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	

XLEN-1	0
x14	
x15	
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	
pc	

There is no dedicated stack pointer or subroutine return address link register in the Base Integer ISA; the instruction encoding allows any `x` *register to be used for these purposes. However, the standard software calling convention uses register* `x1` *to hold the return address for a call, with register* `x5` *available as an alternate link register. The standard calling convention uses register* `x2` *as the stack pointer.*

Hardware might choose to accelerate function calls and returns that use `x1` *or* `x5`*. See the descriptions of the JAL and JALR instructions.*

The optional compressed 16-bit instruction format is designed around the assumption that `x1` *is the return address register and* `x2` *is the stack pointer. Software using other conventions will operate correctly but may have greater code size.*

The number of available architectural registers can have large impacts on code size, performance, and energy consumption. Although 16 registers would arguably be sufficient for an integer ISA running compiled code, it is impossible to encode a complete ISA with 16 registers in 16-bit instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. We wanted to avoid intermediate instruction sizes (such as Xtensa's 24-bit instructions) to simplify base hardware implementations, and once a 32-bit instruction size was adopted, it was straightforward to support 32 integer registers. A larger number of integer registers also helps performance on high-performance code, where there can be extensive use of loop unrolling, software pipelining, and cache tiling.

For these reasons, we chose a conventional size of 32 integer registers for RV32I. Dynamic register usage tends to be dominated by a few frequently accessed registers, and register file implementations can be optimized to reduce access energy for the frequently accessed registers (Tseng & Asanović, 2000). The optional compressed 16-bit instruction format mostly only accesses 8 registers and hence can provide a dense instruction encoding, while additional instruction-set extensions could support a much larger register space (either flat or

hierarchical) if desired.

For resource-constrained embedded applications, we have defined the RV32E subset, which only has 16 registers (Chapter 3).

2.2. Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Base instruction formats. All are a fixed 32 bits in length. The base ISA has `IALIGN=32`, meaning that instructions must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not `IALIGN-bit` aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

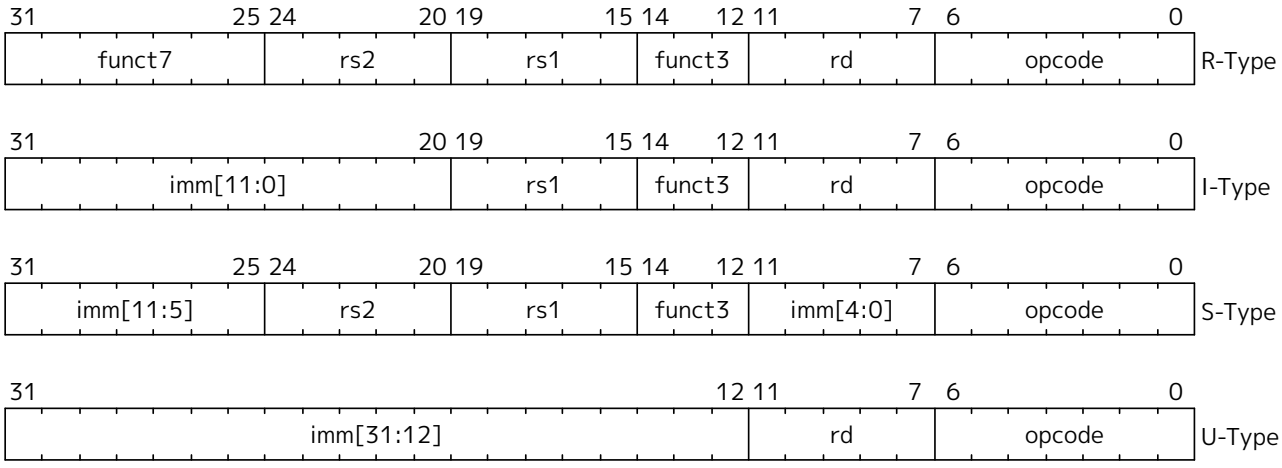
The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added (i.e., IALIGN=16).

Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with IALIGN=32, where these are the only places where misalignment can occur.

The behavior upon decoding a reserved instruction is UNSPECIFIED.

Some platforms may require that opcodes reserved for standard use raise an illegal-instruction exception. Other platforms may permit reserved opcode space be used for non-conforming extensions.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Chapter 6), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.



RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-

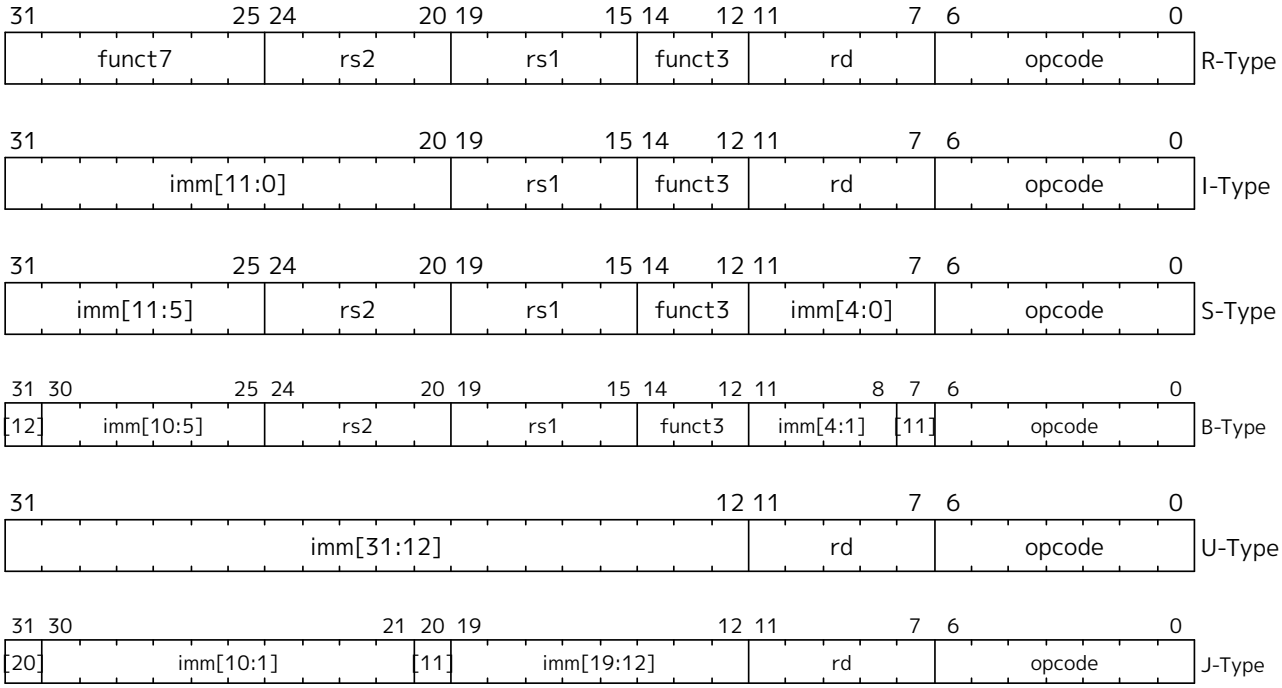
IV aka. SPUR (Lee et al., 1989)).

In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load-upper-immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

Immediates are sign-extended because we did not observe a benefit to using zero extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.

2.3. Immediate Encoding Variants

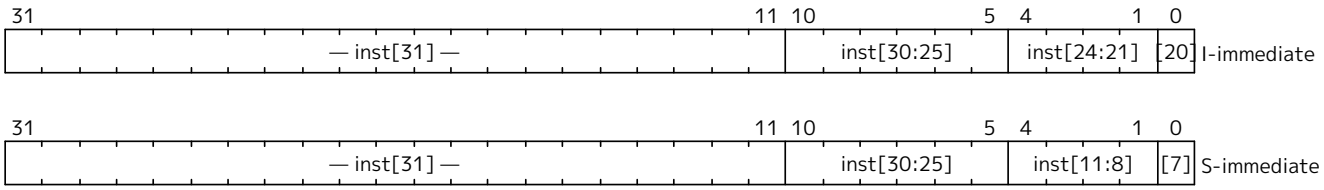
There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Base instruction formats immediate variants..



The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

Immediate types shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[*y*]) produces each bit of the immediate value.



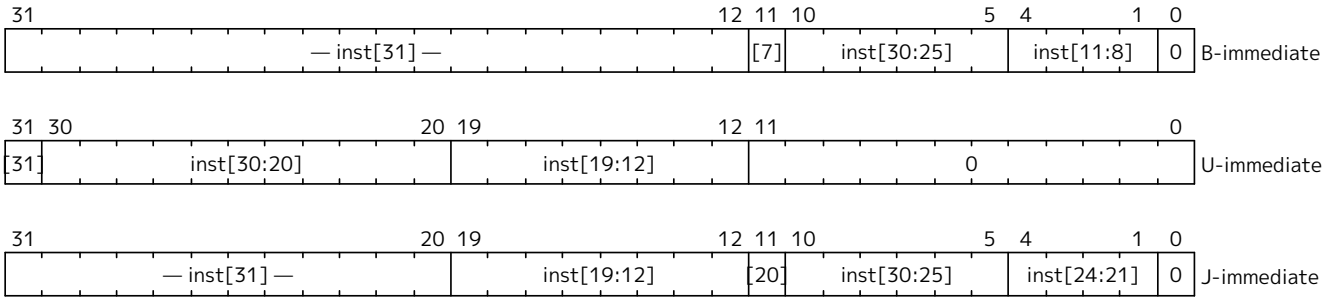


Figure 1. Types of immediate produced by RISC-V instructions.

The fields are labeled with the instruction bits used to construct their value. Sign extensions always uses inst[31].

Sign extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign extension to proceed in parallel with instruction decoding.

Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across types of instruction, we wanted to reduce the hardware cost of the simplest implementations. By rotating bits in the instruction encoding of B and J immediates instead of using dynamic hardware multiplexers to multiply the immediate by 2, we reduce instruction signal fanout and immediate multiplexer costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

2.4. Integer Computational Instructions

Most integer computational instructions operate on `XLEN` bits of values held in the integer register file. Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. The destination is register *rd* for both register-immediate and register-register instructions. No integer computational instructions cause arithmetic exceptions.

We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.

For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. *This covers the common case of addition with an immediate operand.*

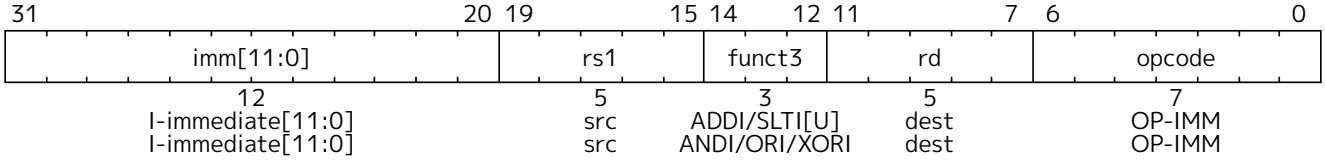
For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
```

```
bne t3, t4, overflow
```

In RV64I, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

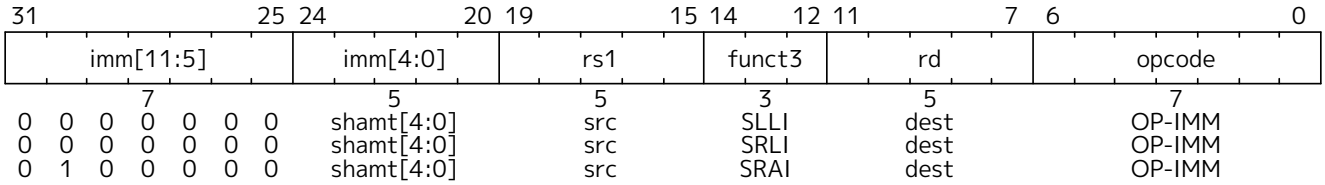
2.4.1. Integer Register-Immediate Instructions



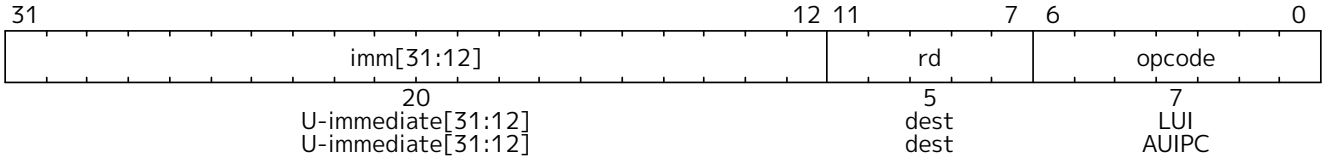
ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the MV *rd, rs1* assembler pseudoinstruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd, rs1, 1* sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudoinstruction SEQZ *rd, rs*).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register *rs1* and the sign-extended 12-bit immediate and place the result in *rd*. Note, XORI *rd, rs1, -1* performs a bitwise logical inversion of register *rs1* (assembler pseudoinstruction NOT *rd, rs*).



Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right-shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).



LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the 32-bit U-immediate value into the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction, then places the result in register *rd*.

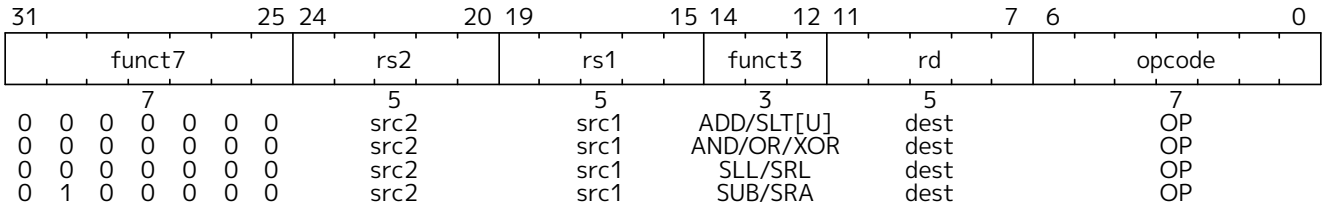
The assembly syntax for `lui` *and* `auipc` *does not represent the lower 12 bits of the U-immediate, which are always zero.*

The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.

The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the local PC (of the instruction following the JAL), it might cause pipeline breaks in simpler microarchitectures or pollute BTB structures in more complex microarchitectures.

2.4.2. Integer Register-Register Instructions

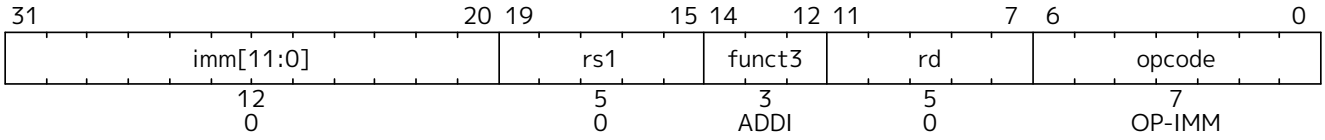
RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

2.4.3. NOP Instruction



The NOP instruction does not change any architecturally visible state, except for advancing the `pc` and incrementing any applicable performance counters. NOP is encoded as ADDI *x0, x0, 0*.

NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as for more readable disassembly output. The other NOP encodings are made available for HINT Instructions.

ADDI was chosen for the NOP encoding as this is most likely to take fewest resources to execute across a range of systems (if not optimized away in decode). In particular, the instruction only reads one register. Also, an ADDI functional unit is more likely to be available in a superscalar design as adds are the most common operation. In particular, address-generation functional units can execute ADDI using the same hardware needed for base+offset address calculations, while register-register ADD or logical/shift operations

require additional hardware.

2.5. Control Transfer Instructions

RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do *not* have architecturally visible delay slots.

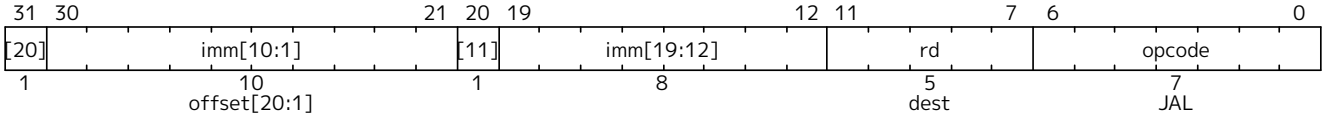
If an instruction access-fault or instruction page-fault exception occurs on the target of a jump or taken branch, the exception is reported on the target instruction, not on the jump or branch instruction.

2.5.1. Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump ('pc'+4) into register *rd*. The standard software calling convention uses 'x1' as the return address register and 'x5' as an alternate link register.

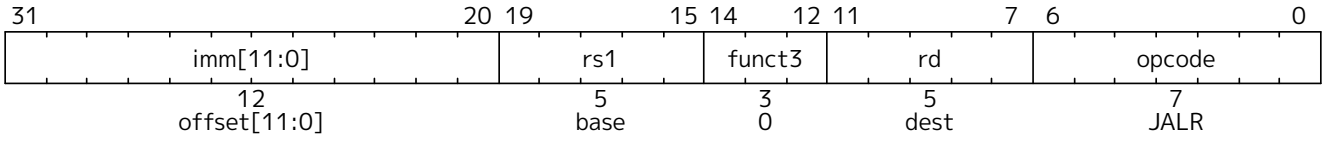
The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register `x5` *was chosen as the alternate link register as it maps to a temporary in the standard calling convention, and has an encoding that is only one bit different than the regular link register.*

Plain unconditional jumps (assembler pseudoinstruction J) are encoded as a JAL with *rd*=`x0`.



The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register *rs1*, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`pc`+4) is written to register *rd*. Register `x0` can be used as the destination if the result is not required.

Plain unconditional indirect jumps (assembler pseudoinstruction JR) are encoded as a JALR with *rd*=`x0`. Procedure returns in the standard calling convention (assembler pseudoinstruction RET) are encoded as a JALR with *rd*=`x0`, *rs1*=`x1`, and *imm*=0.



The unconditional jump instructions all use PC-relative addressing to help support position-independent code. The JALR instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit absolute address range. A LUI instruction can first load rs1 with the upper 20 bits of a target address, then JALR can add in the lower bits. Similarly, AUIPC then JALR can jump anywhere in a 32-bit `pc`*-relative address range.*

Note that the JALR instruction does not treat the 12-bit immediate as multiples of 2 bytes, unlike the conditional branch instructions. This avoids one more immediate format in hardware. In practice, most uses of JALR will have either a zero immediate or be paired with a LUI or AUIPC, so the slight reduction in range is not significant.

Clearing the least-significant bit when calculating the JALR target address both simplifies the hardware slightly and allows the low bit of function pointers to be used to store auxiliary information. Although there is potentially a slight loss of error checking in this case, in practice jumps to an incorrect instruction address will usually quickly raise an exception.

When used with a base rs1=`x0`*, JALR can be used to implement a single instruction subroutine call to the lowest or highest address region from anywhere in the address space, which could be used to implement fast calls to a small runtime library. Alternatively, an ABI could dedicate a general-purpose register to point to a library elsewhere in the address space.*

The JAL and JALR instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.

Instruction-address-misaligned exceptions are not possible on machines with IALIGN=16, e.g., those with the compressed instruction-set extension, C.

Return-address prediction stacks are a common feature of high-performance instruction-fetch units, but require accurate detection of instructions used for procedure calls and returns to be effective. For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. A JAL instruction should push the return address onto a return-address stack (RAS) only when *rd* is 'x1' or `x5`. JALR instructions should push/pop a RAS as shown in Table 3.

Table 3. Return-address stack prediction hints encoded in the register operands of a JALR instruction.

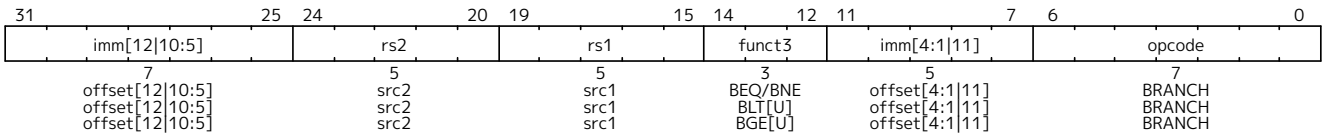
rd is *x1/x5*	*rs1* is *x1/x5*	*rd=rs1*	RAS action
No	No	—	None
No	Yes	—	Pop
Yes	No	—	Push
Yes	Yes	No	Pop, then push
Yes	Yes	Yes	Push

Some other ISAs added explicit hint bits to their indirect-jump instructions to guide return-address stack manipulation. We use implicit hinting tied to register numbers and the calling convention to reduce the encoding space used for these hints.

When two different link registers (`x1` *and* `x5`*) are given as rs1 and rd, then the RAS is both popped and pushed to support coroutines. If rs1 and rd are the same link register (either* `x1` *or* `x5`*), the RAS is only pushed to enable macro-op fusion of the sequences:* `lui ra, imm20; jalr ra, imm12(ra)_ and _auipc ra, imm20; jalr ra, imm12(ra)`

2.5.2. Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes. The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.



Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed

and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

Signed array bounds may be checked with a single BLTU instruction, since any negative index will compare greater than any nonnegative bound.

Software should be optimized such that the sequential code path is the most common path, with less-frequently taken code paths placed out of line. Software should also assume that backward branches will be predicted taken and forward branches as not taken, at least the first time they are encountered. Dynamic predictors should quickly learn any predictable branch behavior.

Unlike some other architectures, the RISC-V jump (JAL with *rd*=`x0`) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always-true condition. RISC-V jumps are also PC-relative and support a much wider offset range than branches, and will not pollute conditional-branch prediction tables.

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC, Xtensa, and MIPS R6), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Another point is that comparisons against zero require non-trivial circuit delay (especially after the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier. There is perhaps an advantage to a design with condition codes in the case where multiple branches can be taken based on the same condition codes, but we believe this case to be relatively rare.

We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs.

We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point operations). Predication adds additional flag state to a system, additional instructions to set and clear flags, and additional encoding overhead on every instruction. Both conditional move and predicated instructions add complexity to out-of-order microarchitectures, adding an implicit third source operand due to the need to copy the original value of the destination architectural register into the renamed destination physical register if the predicate is false. Also, static compile-time decisions to use predication instead of branches can result in lower performance on inputs not included in the compiler training set, especially given that unpredictable branches are rare, and becoming rarer as branch prediction techniques improve.

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict (Heil & Smith, 1996), (Klauser et al., 1998), (Kim et al., 2005) and have been implemented in commercial processors (Sinharoy et al., 2011). The simplest techniques just reduce the penalty of recovering from a mispredicted short forward branch by only flushing instructions in the branch shadow instead of the entire fetch pipeline, or by fetching instructions from both sides using wide instruction fetch or idle instruction fetch slots. More

complex techniques for out-of-order cores add internal predicates on instructions in the branch shadow, with the internal predicate value written by the branch instruction, allowing the branch and following instructions to be executed speculatively and out-of-order with respect to other code.

The conditional branch instructions will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true. If the branch condition evaluates to false, the instruction-address-misaligned exception will not be raised.

Instruction-address-misaligned exceptions are not possible on machines with IALIGN=16, e.g., those with the compressed instruction-set extension, C.

2.6. Load and Store Instructions

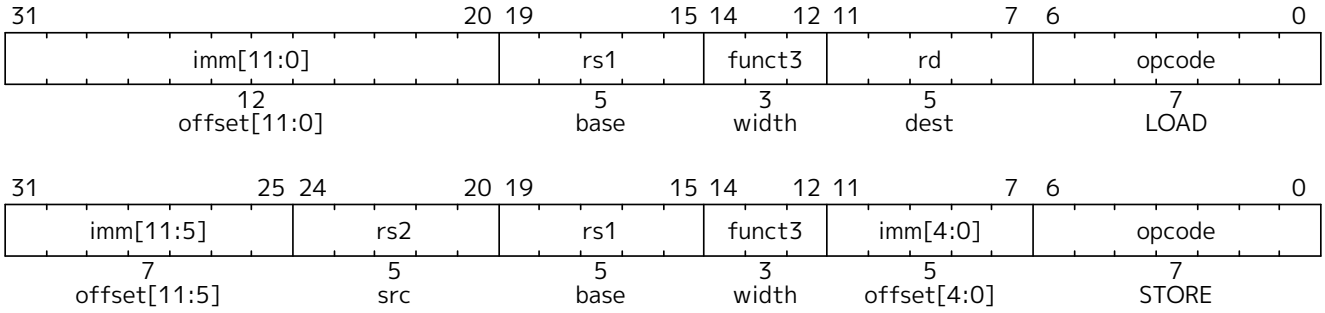
RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed. The EEI will define what portions of the address space are legal to access with which instructions (e.g., some addresses might be read only, or support word access only). Loads with a destination of `x0` must still raise any exceptions and cause any other side effects even though the load value is discarded.

The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.

In a system for which endianness is byte-address invariant, the following property holds: if a byte is stored to memory at some address in some endianness, then a byte-sized load from that address in any endianness returns the stored value.

In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.

In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.



Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB

instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory.

Regardless of EEI, loads and stores whose effective addresses are naturally aligned shall not raise an address-misaligned exception. Loads and stores whose effective address is not naturally aligned to the referenced datatype (i.e., the effective address is not divisible by the size of the access in bytes) have behavior dependent on the EEI.

An EEI may guarantee that misaligned loads and stores are fully supported, and so the software running inside the execution environment will never experience a contained or fatal address-misaligned trap. In this case, the misaligned loads and stores can be handled in hardware, or via an invisible trap into the execution environment implementation, or possibly a combination of hardware and invisible trap depending on address.

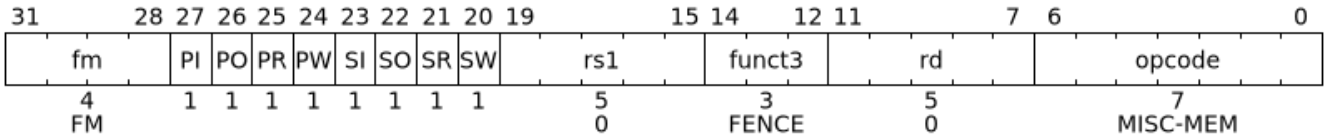
An EEI may not guarantee misaligned loads and stores are handled invisibly. In this case, loads and stores that are not naturally aligned may either complete execution successfully or raise an exception. The exception raised can be either an address-misaligned exception or an access-fault exception. For a memory access that would otherwise be able to complete except for the misalignment, an access-fault exception can be raised instead of an address-misaligned exception if the misaligned access should not be emulated, e.g., if accesses to the memory region have side effects. When an EEI does not guarantee misaligned loads and stores are handled invisibly, the EEI must define if exceptions caused by address misalignment result in a contained trap (allowing software running inside the execution environment to handle the trap) or a fatal trap (terminating execution).

Misaligned accesses are occasionally required when porting legacy code, and help performance on applications when using any form of packed-SIMD extension or handling externally packed data structures. Our rationale for allowing EEIs to choose to support misaligned accesses via the regular load and store instructions is to simplify the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISAs and then provide some separate ISA support for misaligned accesses, either special instructions to help software handle misaligned accesses or a new hardware addressing mode for misaligned accesses. Special instructions are difficult to use, complicate the ISA, and often add new processor state (e.g., SPARC VIS align address offset register) or complicate access to existing processor state (e.g., MIPS LWL/LWR partial register writes). In addition, for loop-oriented packed-SIMD code, the extra overhead when operands are misaligned motivates software to provide multiple forms of loop depending on operand alignment, which complicates code generation and adds to loop startup overhead. New misaligned hardware addressing modes take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).

Even when misaligned loads and stores complete successfully, these accesses might run extremely slowly depending on the implementation (e.g., when implemented via an invisible trap). Furthermore, whereas naturally aligned loads and stores are guaranteed to execute atomically, misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

We do not mandate atomicity for misaligned accesses so execution environment implementations can use an invisible machine trap and a software handler to handle some or all misaligned accesses. If hardware misaligned support is provided, software can exploit this by simply using regular load and store instructions. Hardware can then automatically optimize accesses depending on whether runtime addresses are aligned.

2.7. Memory Ordering Instructions



FENCE instructions are used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a FENCE before any operation in the *predecessor* set preceding the FENCE. Chapter 18 provides a precise description of the RISC-V memory consistency model.

FENCE instructions also order memory reads and writes made by the hart as observed by memory reads and writes made by an external device. However, FENCE instructions do not order observations of events made by an external device using any other signaling mechanism.

A device might observe an access to a memory location via some external communication mechanism, e.g., a memory-mapped control register that drives an interrupt signal to an interrupt controller. This communication is outside the scope of the FENCE ordering mechanism and hence FENCE instructions can provide no guarantee on when a change in the interrupt signal is visible to the interrupt controller. Specific devices might provide additional ordering guarantees to reduce software overhead but those are outside the scope of the RISC-V memory model.

The EEI will define what I/O operations are possible, and in particular, which memory addresses when accessed by load and store instructions will be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new I/O instructions that will also be ordered using the I and O bits in a FENCE instruction.

Table 4. Fence mode encoding

fm field	Mnemonic suffix	Meaning
0000	*none*	Normal Fence
1000	.TSO	With `FENCE RW,RW`: exclude write-to-read ordering; otherwise: *Reserved for future use.*
other	*other*	*Reserved for future use.*

The FENCE mode field *fm* defines the semantics of the FENCE instruction. A `FENCE` (with *fm*=`0000`) orders all memory operations in its predecessor set before all memory operations in its successor set.

A `FENCE.TSO` instruction is encoded as a FENCE instruction with *fm*=`1000`, *predecessor*=`RW`, and *successor* =`RW`. `FENCE.TSO` orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves `non-AMO` store operations in the `FENCE.TSO's` predecessor set unordered with `non-AMO` loads in its successor set.

Because `FENCE RW,RW` *imposes a superset of the orderings that* `FENCE.TSO` *imposes, it is correct to ignore the fm field and implement* `FENCE.TSO` *as* `FENCE RW,RW`.

The unused fields in the FENCE instructions--*rs1* and *rd*--are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard

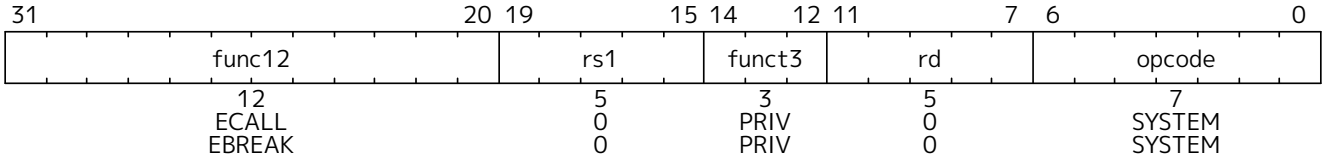
software shall zero these fields. Likewise, many *fm* and predecessor/successor set settings are also reserved for future use. Base implementations shall treat all such reserved configurations as `FENCE` instructions (with *fm*=`0000`), and standard software shall use only non-reserved configurations.

We chose a relaxed memory model to allow high performance from simple machine implementations and from likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative non-memory paths to control added coprocessors or I/O devices. Simple implementations may additionally ignore the predecessor and successor fields and always execute a conservative FENCE on all operations.

2.8. Environment Call and Breakpoints

`SYSTEM` instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in Chapter 6, and the base unprivileged instructions are described in the following section.

The SYSTEM instructions are defined to allow simpler implementations to always trap to a single software trap handler. More sophisticated implementations might execute more of each system instruction in hardware.



These two instructions cause a precise requested trap to the supporting execution environment.

The `ECALL` instruction is used to make a service request to the execution environment. The `EEI` will define how parameters for the service request are passed, but usually these will be in defined locations in the integer register file.

The `EBREAK` instruction is used to return control to a debugging environment.

ECALL and EBREAK were previously named SCALL and SBREAK. The instructions have the same functionality and encoding, but were renamed to reflect that they can be used more generally than to call a supervisor-level operating system or debugger.

EBREAK was primarily designed to be used by a debugger to cause execution to stop and fall back into the debugger. EBREAK is also used by the standard GCC compiler to mark code paths that should not be executed.

Another use of EBREAK is to support "semihosting", where the execution environment includes a debugger that can provide services over an alternate system call interface built around the EBREAK instruction. Because the RISC-V base ISAs do not provide more than one EBREAK instruction, RISC-V semihosting uses a special sequence of instructions to distinguish a semihosting EBREAK from a debugger inserted EBREAK.

```
slli x0, x0, 0x1f   # Entry NOP
ebreak              # Break to debugger
```

```
    srai x0, x0, 7      # Exit NOP
```

Note that these three instructions must be 32-bit-wide instructions, i.e., they mustn't be among the compressed 16-bit instructions described in Chapter 28.

The shift NOP instructions are still considered available for use as HINTs.

Semihosting is a form of service call and would be more naturally encoded as an ECALL using an existing ABI, but this would require the debugger to be able to intercept ECALLs, which is a newer addition to the debug standard. We intend to move over to using ECALLs with a standard ABI, in which case, semihosting can share a service ABI with an existing standard.

We note that ARM processors have also moved to using SVC instead of BKPT for semihosting calls in newer designs.

2.9. HINT Instructions

RV32I reserves a large encoding space for HINT instructions, which are usually used to communicate performance hints to the microarchitecture. Like the NOP instruction, HINTs do not change any architecturally visible state, except for advancing the `pc` and any applicable performance counters. Implementations are always allowed to ignore the encoded hints.

Most RV32I HINTs are encoded as integer computational instructions with *rd*=`x0`. The other RV32I HINTs are encoded as FENCE instructions with a null predecessor or successor set and with *fm*=0.

These HINT encodings have been chosen so that simple implementations can ignore HINTs altogether, and instead execute a HINT as a regular instruction that happens not to mutate the architectural state. For example, ADD is a HINT if the destination register is `x0`; the five-bit rs1 and rs2 fields encode arguments to the HINT. However, a simple implementation can simply execute the HINT as an ADD of rs1 and rs2 that writes `x0`, which has no architecturally visible effect.

As another example, a FENCE instruction with a zero pred field and a zero fm field is a HINT; the succ, rs1, and rd fields encode the arguments to the HINT. A simple implementation can simply execute the HINT as a FENCE that orders the null set of prior memory accesses before whichever subsequent memory accesses are encoded in the succ field. Since the intersection of the predecessor and successor sets is null, the instruction imposes no memory orderings, and so it has no architecturally visible effect.

Table 5 lists all RV32I HINT code points. 91% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs: no standard HINTs will ever be defined in this subspace.

We anticipate standard hints to eventually include memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation.

Table 5. RV32I HINT instructions.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=`x0`	2^{20}	
AUIPC	*rd*=`x0`	2^{20}	
ADDI	*rd*=`x0`, and either *rs1*≠`x0` or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=`x0`	2^{17}	***Designated for future standard use***
ORI	*rd*=`x0`	2^{17}	
XORI	*rd*=`x0`	2^{17}	
ADD	*rd*=`x0`, *rs1*≠`x0`	$2^{10}-32$	
ADD	*rd*=`x0`, *rs1*=`x0`, *rs2*≠`x2-x5`	28	
ADD	*rd*=`x0`, *rs1*=`x0`, *rs2*=`x2-x5`	4	(*rs2*=`x2`) NTL.P1 (*rs2*=`x3`) NTL.PALL (*rs2*=`x4`) NTL.S1 (*rs2*=`x5`) NTL.ALL
SLLI	*rd*=`x0`, *rs1*=`x0`, *shamt*=31	1	Semihosting entry marker
SRAI	*rd*=`x0`, *rs1*=`x0`, *shamt*=7	1	Semihosting exit marker
SUB	*rd*=`x0`	2^{10}	
AND	*rd*=`x0`	2^{10}	
OR	*rd*=`x0`	2^{10}	
XOR	*rd*=`x0`	2^{10}	
SLL	*rd*=`x0`	2^{10}	
SRL	*rd*=`x0`	2^{10}	
SRA	*rd*=`x0`	2^{10}	***Designated for future standard use***
FENCE	*rd*=`x0`, *rs1*≠`x0`, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*≠`x0`, *rs1*=`x0`, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*=0, *succ*≠0	15	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*≠W, *succ*=0	15	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*=W, *succ*=0	1	PAUSE
SLTI	*rd*=`x0`	2^{17}	
SLTIU	*rd*=`x0`	2^{17}	
SLLI	*rd*=`x0`, and either *rs1*≠`x0` or *shamt*≠31	$2^{10}-1$	
SRLI	*rd*=`x0`	2^{10}	***Designated for custom use***
SRAI	*rd*=`x0`, and either *rs1*≠`x0` or *shamt*≠7	$2^{10}-1$	
SLT	*rd*=`x0`	2^{10}	
SLTU	*rd*=`x0`	2^{10}	

`slli x0, x0, 0x1f` *and* `srai x0, x0, 7` *were previously designated as custom HINTs, but they have been appropriated for use in semihosting calls, as described in Section 2.8. To reflect their usage in practice, the base ISA spec has been changed to designate them as standard HINTs.*

Chapter 3. RV32E and RV64E Base Integer Instruction Sets, Version 2.0

This chapter describes the RV32E and RV64E base integer instruction sets, designed for microcontrollers in embedded systems. RV32E and RV64E are reduced versions of RV32I and RV64I, respectively: the only change is to reduce the number of integer registers to 16. This chapter only outlines the differences between RV32E/RV64E and RV32I/RV64I, and so should be read after Chapter 2 and Chapter 4.

RV32E was designed to provide an even smaller base core for embedded microcontrollers. There is also interest in RV64E for microcontrollers within large SoC designs, and to reduce context state for highly threaded 64-bit processors.

Unless otherwise stated, standard extensions compatible with RV32I and RV64I are also compatible with RV32E and RV64E, respectively.

3.1. RV32E and RV64E Programmers' Model

RV32E and RV64E reduce the integer register count to 16 general-purpose registers, (`x0-x15`), where `x0` is a dedicated zero register.

We have found that in the small RV32I core implementations, the upper 16 registers consume around one quarter of the total area of the core excluding memories, thus their removal saves around 25% core area with a corresponding core power reduction.

3.2. RV32E and RV64E Instruction Set Encoding

RV32E and RV64E use the same instruction-set encoding as RV32I and RV64I respectively, except that only registers `x0-x15` are provided. All encodings specifying the other registers `x16-x31` are reserved.

The previous draft of this chapter made all encodings using the `x16-x31` registers available as custom. This version takes a more conservative approach, making these reserved so that they can be allocated between custom space or new standard encodings at a later date.

Chapter 4. RV64I Base Integer Instruction Set, Version 2.1

This chapter describes the RV64I base integer instruction set, which builds upon the RV32I variant described in Chapter 2. This chapter presents only the differences with RV32I, so should be read in conjunction with the earlier chapter.

4.1. Register State

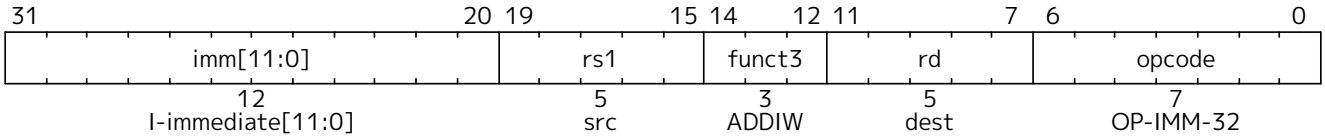
RV64I widens the integer registers and supported user address space to 64 bits (XLEN=64 in Table 2).

4.2. Integer Computational Instructions

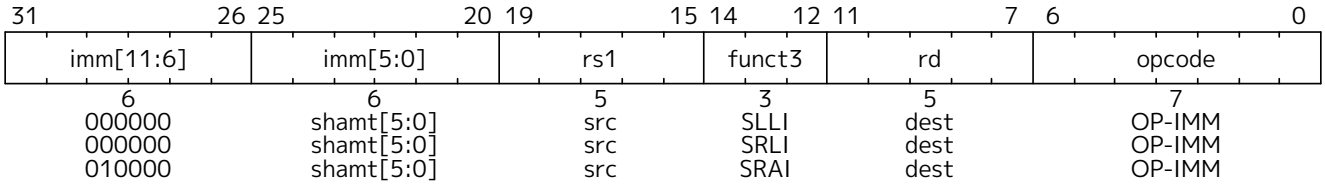
Most integer computational instructions operate on XLEN-bit values. Additional instruction variants are provided to manipulate 32-bit values in RV64I, indicated by a 'W' suffix to the opcode. These "*W" instructions ignore the upper 32 bits of their inputs and always produce 32-bit signed values, sign-extending them to 64 bits, i.e. bits XLEN-1 through 31 are equal.

The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion from a signed 32-bit integer to a signed 64-bit integer. Existing 64-bit wide SLTU and unsigned branch compares still operate correctly on unsigned 32-bit integers under this invariant. Similarly, existing 64-bit wide logical operations on 32-bit sign-extended integers preserve the sign-extension property. A few new instructions (ADD[I]W/SUBW/SxxW) are required for addition and shifts to ensure reasonable performance for 32-bit values.

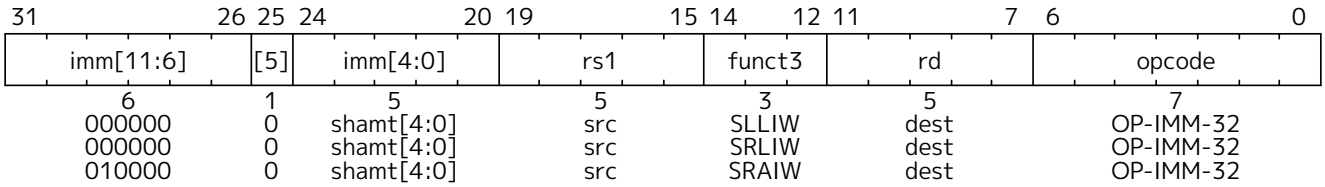
4.2.1. Integer Register-Immediate Instructions



ADDIW is an RV64I instruction that adds the sign-extended 12-bit immediate to register *rs1* and produces the proper sign extension of a 32-bit result in *rd*. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW *rd, rs1, 0* writes the sign extension of the lower 32 bits of register *rs1* into register *rd* (assembler pseudoinstruction SEXT.W).

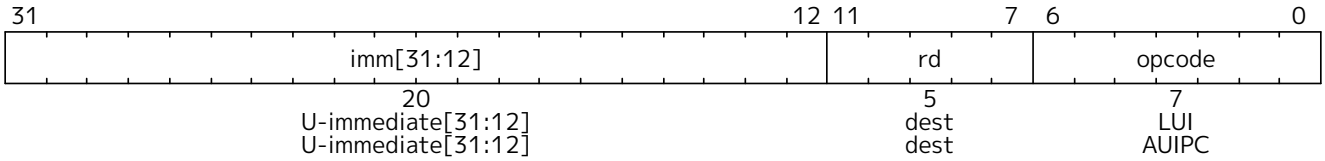


Shifts by a constant are encoded as a specialization of the I-type format using the same instruction opcode as RV32I. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the I-immediate field for RV64I. The right-shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).



SLLIW, SRLIW, and SRAIW are RV64I-only instructions that are analogously defined but operate on 32-bit values and sign-extend their 32-bit results to 64 bits. SLLIW, SRLIW, and SRAIW encodings with *imm[5]* ≠ *0* are reserved.

Previously, SLLIW, SRLIW, and SRAIW with imm[5] ≠ 0 were defined to cause illegal-instruction exceptions, whereas now they are marked as reserved. This is a backwards-compatible change.

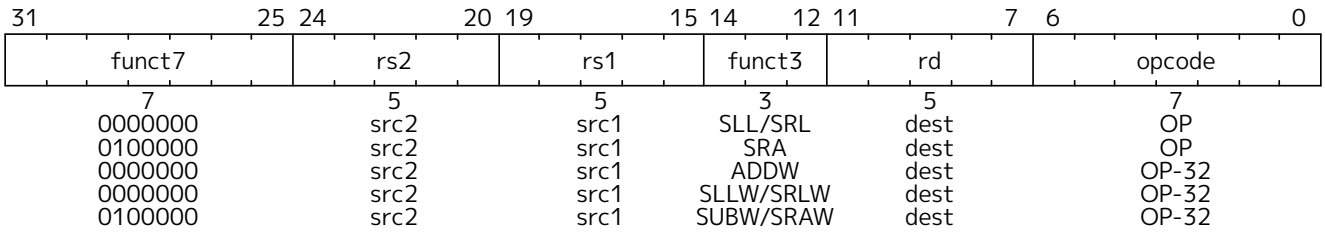


LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 32-bit U-immediate into register *rd*, filling in the lowest 12 bits with zeros. The 32-bit result is sign-extended to 64 bits.

AUIPC (add upper immediate to `pc`) uses the same opcode as RV32I. AUIPC is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the U-immediate, filling in the lowest 12 bits with zeros, sign-extends the result to 64 bits, adds it to the address of the AUIPC instruction, then places the result in register *rd*.

Note that the set of address offsets that can be formed by pairing LUI with LD, AUIPC with JALR, etc. in RV64I is $[-2^{31}-2^{11}, 2^{31}-2^{11}-1]$.

4.2.2. Integer Register-Register Operations



ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

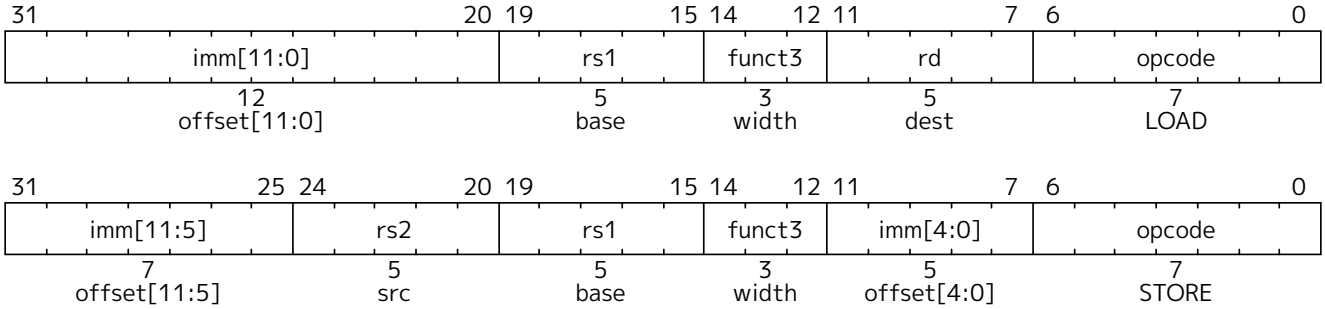
SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in register *rs2*. In RV64I, only the low 6 bits of *rs2* are considered for the shift amount.

SLLW, SRLW, and SRAW are RV64I-only instructions that are analogously defined but operate on 32-bit values and sign-extend their 32-bit results to 64 bits. The shift amount is given by *rs2[4:0]*.

4.3. Load and Store Instructions

RV64I extends the address space to 64 bits. The execution environment will define what portions of the

address space are legal to access.



The LD instruction loads a 64-bit value from memory into register *rd* for RV64I.

The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register *rd* for RV64I. The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64I. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory respectively.

4.4. HINT Instructions

All instructions that are microarchitectural HINTs in RV32I (see Chapter 2) are also HINTs in RV64I.# The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.

Table 6 lists all RV64I HINT code points. 91% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs; no standard HINTs will ever be defined in this subspace.

Table 6. RV64I HINT instructions.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=`x0`	2^{20}	*Designated for future standard use*
AUIPC	*rd*=`x0`	2^{20}	
ADDI	*rd*=`x0`, and either *rs1*≠`x0` or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=`x0`	2^{17}	
ORI	*rd*=`x0`	2^{17}	
XORI	*rd*=`x0`	2^{17}	
ADDIW	*rd*=`x0`	2^{17}	
ADD	*rd*=`x0`, *rs1*≠`x0`	$2^{10}-32$	
ADD	*rd*=`x0`, *rs1*=`x0`, *rs2*≠_x2_-*x5*	28	
ADD	*rd*=`x0`, *rs1*=`x0`, *rs2=x2-x5*	4	(*rs2=x2*) NTL.P1 (*rs2=x3*) NTL.PALL (*rs2=x4*) NTL.S1 (*rs2=x5*) NTL.ALL
SLLI	*rd*=`x0`, *rs1*=`x0`, *shamt*=31	1	Semihosting entry marker
SRAI	*rd*=`x0`, *rs1*=`x0`, *shamt*=7	1	Semihosting exit marker

Instruction	Constraints	Code Points	Purpose
SUB	*rd*=`x0`	2^{10}	*Designated for future standard use*
AND	*rd*=`x0`	2^{10}	
OR	*rd*=`x0`	2^{10}	
XOR	*rd*=`x0`	2^{10}	
SLL	*rd*=`x0`	2^{10}	
SRL	*rd*=`x0`	2^{10}	
SRA	*rd*=`x0`	2^{10}	
ADDW	*rd*=`x0`	2^{10}	
SUBW	*rd*=`x0`	2^{10}	
SLLW	*rd*=`x0`	2^{10}	
SRLW	*rd*=`x0`	2^{10}	
SRAW	*rd*=`x0`	2^{10}	
FENCE	*rd*=`x0`, *rs1*≠`x0`, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*≠`x0`, *rs1*=`x0`, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10}-63$	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*=0, *succ*≠0	15	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*≠W, *succ*=0	15	
FENCE	*rd*=*rs1*=`x0`, *fm*=0, *pred*=W, *succ*=0	1	PAUSE
SLTI	*rd*=`x0`	2^{17}	*Designated for custom use*
SLTIU	*rd*=`x0`	2^{17}	
SLLI	*rd*=`x0`, and either *rs1*≠`x0` or *shamt*≠31	$2^{11}-1$	
SRLI	*rd*=`x0`	2^{11}	
SRAI	*rd*=`x0`, and either *rs1*≠`x0` or *shamt*≠7	$2^{11}-1$	
SLLIW	*rd*=`x0`	2^{10}	
SRLIW	*rd*=`x0`	2^{10}	
SRAIW	*rd*=`x0`	2^{10}	
SLT	*rd*=`x0`	2^{10}	
SLTU	*rd*=`x0`	2^{10}	

`slli x0, x0, 0x1f` *and* `srai x0, x0, 7` *were previously designated as custom HINTs, but they have been appropriated for use in semihosting calls, as described in Section 2.8. To reflect their usage in practice, the base ISA spec has been changed to designate them as standard HINTs.*

Chapter 5. "Zifencei" Extension for Instruction-Fetch Fence, Version 2.0

This chapter defines the "Zifencei" extension, which includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart. Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetches.

We considered but did not include a "store instruction word" instruction as in (Tremblay et al., 2000). JIT compilers may generate a large trace of instructions before a single FENCE.I, and amortize any instruction cache snooping/invalidation overhead by writing translated instructions to memory regions that are known not to reside in the I-cache.

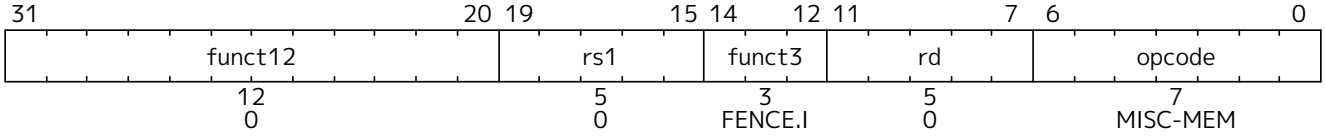
The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, or if the memory system consists of only uncached RAMs, then just the fetch pipeline needs to be flushed at a FENCE.I.

The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.

First, it has been recognized that on some systems, FENCE.I will be expensive to implement and alternate mechanisms are being discussed in the memory model task group. In particular, for designs that have an incoherent instruction cache and an incoherent data cache, or where the instruction cache refill does not snoop a coherent data cache, both caches must be completely flushed when a FENCE.I instruction is encountered. This problem is exacerbated when there are multiple levels of I and D cache in front of a unified cache or outer memory system.

Second, the instruction is not powerful enough to make available at user level in a Unix-like operating system environment. The FENCE.I only synchronizes the local hart, and the OS can reschedule the user hart to a different physical hart after the FENCE.I. This would require the OS to execute an additional FENCE.I as part of every context migration. For this reason, the standard Linux ABI has removed FENCE.I from user-level and now requires a system call to maintain instruction-fetch coherence, which allows the OS to minimize the number of FENCE.I executions required on current systems and provides forward-compatibility with future improved instruction-fetch coherence mechanisms.

Future approaches to instruction-fetch coherence under discussion include providing more restricted versions of FENCE.I that only target a given address specified in rs1, and/or allowing software to use an ABI that relies on machine-mode cache-maintenance operations.



The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction. A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does *not* ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart also has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.

A FENCE.I instruction orders all explicit memory accesses that precede the FENCE.I in program order before all instruction fetches that follow the FENCE.I in program order.

In the following litmus test, for example, the outcome `a0`*=1,* `a1`*=0 on the consumer hart is forbidden, assuming little-endian RV32IC harts:*

```
Initially, flag = 0.

Producer hart:                                  Consumer hart:

la t0, patch_me                                 la t2, flag
li t1, 0x4585                                   lw a0, (t2)
sh t1, (t0)   # patch_me := c.li a1, 1          fence.i
fence w, w    # order flag write              patch_me:
la t0, flag                                     c.li a1, 0
li t1, 1
sw t1, (t0)   # flag := 1
```

Note that this example is only meant to illustrate the aforementioned ordering property. In a realistic producer-consumer code-generation scheme, the consumer would loop until `flag` *becomes 1 before executing the FENCE.I instruction.*

An instruction fetch is always ordered before any explicit memory accesses that instruction gives rise to.

The unused fields in the FENCE.I instruction, *funct12*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

Because FENCE.I only orders stores with a hart's own instruction fetches, application code should only rely upon FENCE.I if the application thread will not be migrated to a different hart. The EEI can provide mechanisms for efficient multiprocessor instruction-stream synchronization.

Chapter 6. "Zicsr" Extension for Control and Status Register (CSR) Instructions, Version 2.0

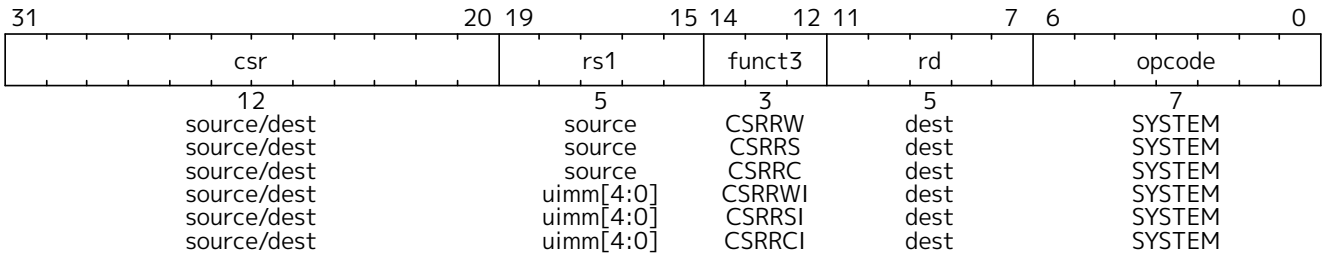
RISC-V defines a separate address space of 4096 Control and Status registers associated with each hart. This chapter defines the full set of CSR instructions that operate on these CSRs.

While CSRs are primarily used by the privileged architecture, there are several uses in unprivileged code including for counters and timers, and for floating-point status.

The counters and timers are no longer considered mandatory parts of the standard base ISAs, and so the CSR instructions required to access them have been moved out of Chapter 2 into this separate chapter.

6.1. CSR Instructions

All CSR instructions atomically read-modify-write a single CSR, whose CSR specifier is encoded in the 12-bit *csr* field of the instruction held in bits 31-20. The immediate forms use a 5-bit zero-extended immediate encoded in the *rs1* field.



The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable.

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable.

Since CSRRS and CSRRC perform a read-modify-write operation, any bits that read as a different value to their underlying value may be modified by these instructions even if the corresponding bit is not set in rs1. For example, pmpaddrn[G-1] may have an underlying value of 1 but read as 0. Executing CSRRC or CSRRS to modify a different bit will cause 0 to be read from pmpaddrn[G-1] and then written back, updating the underlying value to 0.

For both CSRRS and CSRRC, if *rs1*=`x0`, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal-instruction exceptions on accesses to read-only CSRs. Both CSRRS and CSRRC always read the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields. Note that if *rs1* specifies a register other than `x0`, and that register holds a zero value, the instruction will not action any attendant per-field side effects, but will action any side effects caused by writing to the entire CSR.

A CSRRW with *rs1*=`x0` will attempt to write zero to the destination CSR.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the *rs1* field instead of a value from an integer register. For CSRRSI and CSRRCI, if the uimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal-instruction exceptions on accesses to read-only CSRs. For CSRRWI, if *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side effects regardless of *rd* and *rs1* fields.

Table 7. Conditions determining whether a CSR instruction reads or writes the specified CSR.

Register operand				
Instruction	*rd* is `x0`	*rs1* is `x0`	Reads CSR	Writes CSR
CSRRW	Yes	-	No	Yes
CSRRW	No	-	Yes	Yes
CSRRS/CSRRC	-	Yes	Yes	No
CSRRS/CSRRC	-	No	Yes	Yes
Immediate operand				
Instruction	*rd* is `x0`	*uimm*=0	Reads CSR	Writes CSR
CSRRWI	Yes	-	No	Yes
CSRRWI	No	-	Yes	Yes
CSRRSI/CSRRCI	-	Yes	Yes	No
CSRRSI/CSRRCI	-	No	Yes	Yes

Table 7 summarizes the behavior of the CSR instructions with respect to whether they read and/or write the CSR.

In addition to side effects that occur as a consequence of reading or writing a CSR, individual fields within a CSR might have side effects when written. The CSRRW[I] instructions action side effects for all such fields within the written CSR. The CSRRS[I] and CSRRC[I] instructions only action side effects for fields for which the *rs1* or *uimm* argument has at least one bit set corresponding to that field.

As of this writing, no standard CSRs have side effects on field writes. Hence, whether a standard CSR access has any side effects can be determined solely from the opcode.

Defining CSRs with side effects on field writes is not recommended.

For any event or consequence that occurs due to a CSR having a particular value, if a write to the CSR gives it that value, the resulting event or consequence is said to be an *indirect effect* of the write. Indirect effects of a CSR write are not considered by the RISC-V ISA to be side effects of that write.

An example of side effects for CSR accesses would be if reading from a specific CSR causes a light bulb to turn on, while writing an odd value to the same CSR causes the light to turn off. Assume writing an even value has no effect. In this case, both the read and write have side effects controlling whether the bulb is lit, as this condition is not determined solely from the CSR value. (Note that after writing an odd value to the CSR to turn off the light, then reading to turn the light on, writing again the same odd value causes the light to turn off again. Hence, on the last write, it is not a change in the CSR value that turns off the light.)

On the other hand, if a bulb is rigged to light whenever the value of a particular CSR is odd,

then turning the light on and off is not considered a side effect of writing to the CSR but merely an indirect effect of such writes.

More concretely, the RISC-V privileged architecture defined in Volume II specifies that certain combinations of CSR values cause a trap to occur. When an explicit write to a CSR creates the conditions that trigger the trap, the trap is not considered a side effect of the write but merely an indirect effect.

Standard CSRs do not have any side effects on reads. Standard CSRs may have side effects on writes. Custom extensions might add CSRs for which accesses have side effects on either reads or writes.

Some CSRs, such as the instructions-retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes such a CSR, the explicit write is done instead of the update from the side effect. In particular, a value written to `instret` by one instruction will be the value read by the following instruction.

The assembler pseudoinstruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudoinstruction to write a CSR, CSRW *csr, rs1*, is encoded as CSRRW *x0, csr, rs1*, while CSRWI *csr, uimm*, is encoded as CSRRWI *x0, csr, uimm*.

Further assembler pseudoinstructions are defined to set and clear bits in the CSR when the old value is not required: CSRS/CSRC *csr, rs1*; CSRSI/CSRCI *csr, uimm*.

6.1.1. CSR Access Ordering

Each RISC-V hart normally observes its own CSR accesses, including its implicit CSR accesses, as performed in program order. In particular, unless specified otherwise, a CSR access is performed after the execution of any prior instructions in program order whose behavior modifies or is modified by the CSR state and before the execution of any subsequent instructions in program order whose behavior modifies or is modified by the CSR state. Furthermore, an explicit CSR read returns the CSR state before the execution of the instruction, while an explicit CSR write suppresses and overrides any implicit writes or modifications to the same CSR by the same instruction.

Likewise, any side effects from an explicit CSR access are normally observed to occur synchronously in program order. Unless specified otherwise, the full consequences of any such side effects are observable by the very next instruction, and no consequences may be observed out-of-order by preceding instructions. (Note the distinction made earlier between side effects and indirect effects of CSR writes.)

For the RVWMO memory consistency model (Chapter 18), CSR accesses are weakly ordered by default, so other harts or devices may observe CSR accesses in an order different from program order. In addition, CSR accesses are not ordered with respect to explicit memory accesses, unless a CSR access modifies the execution behavior of the instruction that performs the explicit memory access or unless a CSR access and an explicit memory access are ordered by either the syntactic dependencies defined by the memory model or the ordering requirements defined by the Memory-Ordering PMAs section in Volume II of this manual. To enforce ordering in all other cases, software should execute a FENCE instruction between the relevant accesses. For the purposes of the FENCE instruction, CSR read accesses are classified as device input (I), and CSR write accesses are classified as device output (O).

Informally, the CSR space acts as a weakly ordered memory-mapped I/O region, as defined by the Memory-Ordering PMAs section in Volume II of this manual. As a result, the order of CSR accesses with respect to all other accesses is constrained by the same mechanisms that constrain the order of memory-mapped I/O accesses to such a region.

> *These CSR-ordering constraints are imposed to support ordering main memory and memory-mapped I/O accesses with respect to CSR accesses that are visible to, or affected by, devices or other harts. Examples include the* `time`, `cycle`, *and* `mcycle` *CSRs, in addition to CSRs that reflect pending interrupts, like* `mip` *and* `sip`*. Note that implicit reads of such CSRs (e.g., taking an interrupt because of a change in* `mip`*) are also ordered as device input.*
>
> *Most CSRs (including, e.g., the* `fcsr`*) are not visible to other harts; their accesses can be freely reordered in the global memory order with respect to FENCE instructions without violating this specification.*

The hardware platform may define that accesses to certain CSRs are strongly ordered, as defined by the Memory-Ordering PMAs section in Volume II of this manual. Accesses to strongly ordered CSRs have stronger ordering constraints with respect to accesses to both weakly ordered CSRs and accesses to memory-mapped I/O regions.

The rules for the reordering of CSR accesses in the global memory order should probably be moved to Chapter 18 concerning the RVWMO memory consistency model.

Chapter 7. "Zicntr" and "Zihpm" Extensions for Counters, Version 2.0

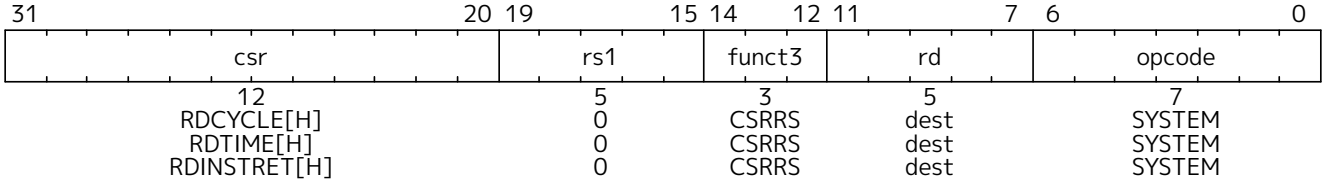
RISC-V ISAs provide a set of up to thirty-two 64-bit performance counters and timers that are accessible via unprivileged XLEN-bit read-only CSR registers `0xC00`–`0xC1F` (when XLEN=32, the upper 32 bits are accessed via CSR registers `0xC80`–`0xC9F`). These counters are divided between the "Zicntr" and "Zihpm" extensions.

7.1. "Zicntr" Extension for Base Counters and Timers

The Zicntr standard extension comprises the first three of these counters (CYCLE, TIME, and INSTRET), which have dedicated functions (cycle count, real-time clock, and instructions retired, respectively). The Zicntr extension depends on the Zicsr extension.

We recommend provision of these basic counters in implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters in the separate Zihpm extension can help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

Some execution environments might prohibit access to counters, for example, to impede timing side-channel attacks.



For base ISAs with XLEN≥64, CSR instructions can access the full 64-bit CSRs directly. In particular, the RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the `cycle`, `time`, and `instret` counters.

The counter pseudoinstructions are mapped to the read-only `csrrs rd, counter, x0` canonical form, but the other read-only CSR instruction forms (based on CSRRC/CSRRSI/CSRRCI) are also legal ways to read these CSRs.

For base ISAs with XLEN=32, the Zicntr extension enables the three 64-bit read-only counters to be accessed in 32-bit pieces. The RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions provide the lower 32 bits, and the RDCYCLEH, RDTIMEH, and RDINSTRETH pseudoinstructions provide the upper 32 bits of the respective counters.

We required the counters be 64 bits wide, even when XLEN=32, as otherwise it is very difficult for software to determine if values have overflowed. The sample code given below shows how the full 64-bit width value can be safely read using the individual 32-bit width pseudoinstructions.

The RDCYCLE pseudoinstruction reads the low XLEN bits of the `cycle` CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. RDCYCLEH is only present when XLEN=32 and reads bits 63-32 of the same cycle counter. The underlying 64-bit counter should never overflow in practice. The rate at which the cycle counter advances will depend on the implementation and operating environment. The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle

counter is incrementing.

RDCYCLE is intended to return the number of cycles executed by the processor core, not the hart. Precisely defining what is a "core" is difficult given some implementation choices (e.g., AMD Bulldozer). Precisely defining what is a "clock cycle" is also difficult given the range of implementations (including software emulations), but the intent is that RDCYCLE is used for performance monitoring along with the other performance counters. In particular, where there is one hart/core, one would expect cycle-count/instructions-retired to measure CPI for a hart.

Cores don't have to be exposed to software at all, and an implementer might choose to pretend multiple harts on one physical core are running on separate cores with one hart/core, and provide separate cycle counters for each hart. This might make sense in a simple barrel processor (e.g., CDC 6600 peripheral processors) where inter-hart timing interactions are non-existent or minimal.

Where there is more than one hart/core and dynamic multithreading, it is not generally possible to separate out cycles per hart (especially with SMT). It might be possible to define a separate performance counter that tried to capture the number of cycles a particular hart was running, but this definition would have to be very fuzzy to cover all the possible threading implementations. For example, should we only count cycles for which any instruction was issued to execution for this hart, and/or cycles any instruction retired, or include cycles this hart was occupying machine resources but couldn't execute due to stalls while other harts went into execution? Likely, "all of the above" would be needed to have understandable performance stats. This complexity of defining a per-hart cycle count, and also the need in any case for a total per-core cycle count when tuning multithreaded code led to just standardizing the per-core cycle counter, which also happens to work well for the common single hart/core case.

Standardizing what happens during "sleep" is not practical given that what "sleep" means is not standardized across execution environments, but if the entire core is paused (entirely clock-gated or powered-down in deep sleep), then it is not executing clock cycles, and the cycle count shouldn't be increasing per the spec. There are many details, e.g., whether clock cycles required to reset a processor after waking up from a power-down event should be counted, and these are considered execution-environment-specific details.

Even though there is no precise definition that works for all platforms, this is still a useful facility for most platforms, and an imprecise, common, "usually correct" standard here is better than no standard. The intent of RDCYCLE was primarily performance monitoring/tuning, and the specification was written with that goal in mind.

The RDTIME pseudoinstruction reads the low XLEN bits of the "time" CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is only present when XLEN=32 and reads bits 63-32 of the same real-time counter. The underlying 64-bit counter increments by one with each tick of the real-time clock, and, for realistic real-time clock frequencies, should never overflow in practice. The execution environment should provide a means of determining the period of a counter tick (seconds/tick). The period should be constant within a small error bound. The environment should provide a means to determine the accuracy of the clock (i.e., the maximum relative error between the nominal and actual real-time clock periods).

On some simple platforms, cycle count might represent a valid implementation of RDTIME, in which case RDTIME and RDCYCLE may return the same result.

It is difficult to provide a strict mandate on clock period given the wide variety of possible implementation platforms. The maximum error bound should be set based on the requirements of the platform.

The real-time clocks of all harts must be synchronized to within one tick of the real-time clock.

As with other architectural mandates, it suffices to appear "as if" harts are synchronized to within one tick of the real-time clock, i.e., software is unable to observe that there is a greater delta between the real-time clock values observed on two harts.

If, for example, the real-time clock increments at a frequency of 1 GHz, then all harts must appear to be synchronized to within 1 nsec. But it is also acceptable for this example implementation to only update the real-time clock at, say, a frequency of 100 MHz with increments of 10 ticks. As long as software cannot observe this seeming violation of the above synchronization requirement, and software always observes time across harts to be monotonically nondecreasing, then this implementation is compliant.

A platform spec may then, for example, specify an apparent real-time clock tick frequency (e.g. 1 GHz) and also a minimum update frequency (e.g. 100 MHz) at which updated time values are guaranteed to be observable by software. Software may read time more frequently, but it should only observe monotonically nondecreasing values and it should observe a new value at least once every 10 ns (corresponding to the 100 MHz update frequency in this example).

The RDINSTRET pseudoinstruction reads the low XLEN bits of the `instret` CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past. RDINSTRETH is only present when XLEN=32 and reads bits 63-32 of the same instruction counter. The underlying 64-bit counter should never overflow in practice.

Instructions that cause synchronous exceptions, including ECALL and EBREAK, are not considered to retire and hence do not increment the `instret` *CSR.*

The following code sequence will read a valid 64-bit cycle counter value into `x3:x2`, even if the counter overflows its lower half between reading its upper and lower halves.

```
again:
    rdcycleh    x3
    rdcycle     x2
    rdcycleh    x4
    bne         x3, x4, again
```

Listing 1. Sample code for reading the 64-bit cycle counter when XLEN=32.

7.2. "Zihpm" Extension for Hardware Performance Counters

The Zihpm extension comprises up to 29 additional unprivileged 64-bit hardware performance counters, `hpmcounter3-hpmcounter31`. When XLEN=32, the upper 32 bits of these performance counters are accessible via additional CSRs `hpmcounter3h- hpmcounter31h`. The Zihpm extension depends on the Zicsr extension.

In some applications, it is important to be able to read multiple counters at the same instant in time. When run under a multitasking environment, a user thread can suffer a context switch while attempting to read the counters. One solution is for the user thread to read the real-time counter before and after reading the other counters to determine if a context switch occurred in the middle of the sequence, in which case the reads can be retried. We considered adding output latches to allow a user thread to snapshot the counter values atomically, but this would increase the size of the user context, especially for implementations with a richer set of counters.

The implemented number and width of these additional counters, and the set of events they count, are platform-specific. Accessing an unimplemented counter may cause an illegal-instruction exception or may return a constant value. If the configuration used to select the events counted by a counter is misconfigured, the counter may return a constant value.

The execution environment should provide a means to determine the number and width of the implemented counters, and an interface to configure the events to be counted by each counter.

For execution environments implemented on RISC-V privileged platforms, the privileged architecture manual describes privileged CSRs controlling access by lower privileged modes to these counters, and to set the events to be counted.

Alternative execution environments (e.g., user-level-only software performance models) may provide alternative mechanisms to configure the events counted by the performance counters.

It would be useful to eventually standardize event settings to count ISA-level metrics, such as the number of floating-point instructions executed for example, and possibly a few common microarchitectural metrics, such as "L1 instruction cache misses".

Chapter 8. "Zihintntl" Extension for Non-Temporal Locality Hints, Version 1.0

The NTL instructions are HINTs that indicate that the explicit memory accesses of the immediately subsequent instruction (henceforth "target instruction") exhibit poor temporal locality of reference. The NTL instructions do not change architectural state, nor do they alter the architecturally visible effects of the target instruction. Four variants are provided:

The NTL.P1 instruction indicates that the target instruction does not exhibit temporal locality within the capacity of the innermost level of private cache in the memory hierarchy. NTL.P1 is encoded as ADD *x0, x0, x2*.

The NTL.PALL instruction indicates that the target instruction does not exhibit temporal locality within the capacity of any level of private cache in the memory hierarchy. NTL.PALL is encoded as ADD *x0, x0, x3*.

The NTL.S1 instruction indicates that the target instruction does not exhibit temporal locality within the capacity of the innermost level of shared cache in the memory hierarchy. NTL.S1 is encoded as ADD *x0, x0, x4*.

The NTL.ALL instruction indicates that the target instruction does not exhibit temporal locality within the capacity of any level of cache in the memory hierarchy. NTL.ALL is encoded as ADD *x0, x0, x5*.

The NTL instructions can be used to avoid cache pollution when streaming data or traversing large data structures, or to reduce latency in producer-consumer interactions.

A microarchitecture might use the NTL instructions to inform the cache replacement policy, or to decide which cache to allocate into, or to avoid cache allocation altogether. For example, NTL.P1 might indicate that an implementation should not allocate a line in a private L1 cache, but should allocate in L2 (whether private or shared). In another implementation, NTL.P1 might allocate the line in L1, but in the least-recently used state.

NTL.ALL will typically inform implementations not to allocate anywhere in the cache hierarchy. Programmers should use NTL.ALL for accesses that have no exploitable temporal locality.

Like any HINTs, these instructions may be freely ignored. Hence, although they are described in terms of cache-based memory hierarchies, they do not mandate the provision of caches.

Some implementations might respect these HINTs for some memory accesses but not others: e.g., implementations that implement LR/SC by acquiring a cache line in the exclusive state in L1 might ignore NTL instructions on LR and SC, but might respect NTL instructions for AMOs and regular loads and stores.

Table 8 lists several software use cases and the recommended NTL variant that *portable* software—i.e., software not tuned for any specific implementation's memory hierarchy—should use in each case.

Table 8. Recommended NTL variant for portable software to employ in various scenarios.

Scenario	Recommended NTL variant
Access to a working set between and in size	NTL.P1
Access to a working set between and in size	NTL.PALL
Access to a working set greater than in size	NTL.S1
Access with no exploitable temporal locality (e.g., streaming)	NTL.ALL

Scenario	Recommended NTL variant
Access to a contended synchronization variable	NTL.PALL

The working-set sizes listed in Table 8 are not meant to constrain implementers' cache-sizing decisions. Cache sizes will obviously vary between implementations, and so software writers should only take these working-set sizes as rough guidelines.

Table 9 lists several sample memory hierarchies and recommends how each NTL variant maps onto each cache level. The table also recommends which NTL variant that implementation-tuned software should use to avoid allocating in a particular cache level. For example, for a system with a private L1 and a shared L2, it is recommended that NTL.P1 and NTL.PALL indicate that temporal locality cannot be exploited by the L1, and that NTL.S1 and NTL.ALL indicate that temporal locality cannot be exploited by the L2. Furthermore, software tuned for such a system should use NTL.P1 to indicate a lack of temporal locality exploitable by the L1, or should use NTL.ALL indicate a lack of temporal locality exploitable by the L2.

If the C or Zca extension is provided, compressed variants of these HINTs are also provided: C.NTL.P1 is encoded as C.ADD *x0, x2*; C.NTL.PALL is encoded as C.ADD *x0, x3*; C.NTL.S1 is encoded as C.ADD *x0, x4*; and C.NTL.ALL is encoded as C.ADD *x0, x5*.

The NTL instructions affect all memory-access instructions except the cache-management instructions in the Zicbom extension.

As of this writing, there are no other exceptions to this rule, and so the NTL instructions affect all memory-access instructions defined in the base ISAs and the A, F, D, Q, C, and V standard extensions, as well as those defined within the hypervisor extension in Volume II.

The NTL instructions can affect cache-management operations other than those in the Zicbom extension. For example, NTL.PALL followed by CBO.ZERO might indicate that the line should be allocated in L3 and zeroed, but not allocated in L1 or L2.

Table 9. Mapping of NTL variants to various memory hierarchies.

Memory hierarchy	Recommended mapping of NTL variant to actual cache level				Recommended NTL variant for explicit cache management			
	P1	PALL	S1	ALL	L1	L2	L3	L4/L5
Common Scenarios								
No caches	---				none			
Private L1 only	L1	L1	L1	L1	ALL	---	---	---
Private L1; shared L2	L1	L1	L2	L2	P1	ALL	---	---
Private L1; shared L2/L3	L1	L1	L2	L3	P1	S1	ALL	---
Private L1/L2	L1	L2	L2	L2	P1	ALL	---	---
Private L1/L2; shared L3	L1	L2	L3	L3	P1	PALL	ALL	---
Private L1/L2; shared L3/L4	L1	L2	L3	L4	P1	PALL	S1	ALL
Uncommon Scenarios								
Private L1/L2/L3; shared L4	L1	L3	L4	L4	P1	P1	PALL	ALL
Private L1; shared L2/L3/L4	L1	L1	L2	L4	P1	S1	ALL	ALL
Private L1/L2; shared L3/L4/L5	L1	L2	L3	L5	P1	PALL	S1	ALL
Private L1/L2/L3; shared L4/L5	L1	L3	L4	L5	P1	P1	PALL	ALL

When an NTL instruction is applied to a prefetch hint in the Zicbop extension, it indicates that a cache line should be prefetched into a cache that is *outer* from the level specified by the NTL.

For example, in a system with a private L1 and shared L2, NTL.P1 followed by PREFETCH.R might prefetch into L2 with read intent.

To prefetch into the innermost level of cache, do not prefix the prefetch instruction with an NTL instruction.

In some systems, NTL.ALL followed by a prefetch instruction might prefetch into a cache or prefetch buffer internal to a memory controller.

Software is discouraged from following an NTL instruction with an instruction that does not explicitly access memory. Nonadherence to this recommendation might reduce performance but otherwise has no architecturally visible effect.

In the event that a trap is taken on the target instruction, implementations are discouraged from applying the NTL to the first instruction in the trap handler. Instead, implementations are recommended to ignore the HINT in this case.

If an interrupt occurs between the execution of an NTL instruction and its target instruction, execution will normally resume at the target instruction. That the NTL instruction is not re-executed does not change the semantics of the program.

Some implementations might prefer not to process the NTL instruction until the target instruction is seen (e.g., so that the NTL can be fused with the memory access it modifies). Such implementations might preferentially take the interrupt before the NTL, rather than between the NTL and the memory access.

Since the NTL instructions are encoded as ADDs, they can be used within LR/SC loops without voiding the forward-progress guarantee. But, since using other loads and stores within an LR/SC loop does void the forward-progress guarantee, the only reason to use an NTL within

such a loop is to modify the LR or the SC.

Chapter 9. "Zihintpause" Extension for Pause Hint, Version 2.0

The PAUSE instruction is a HINT that indicates the current hart's rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero.

Software can use the PAUSE instruction to reduce energy consumption while executing spin-wait code sequences. Multithreaded cores might temporarily relinquish execution resources to other harts when PAUSE is executed. It is recommended that a PAUSE instruction generally be included in the code sequence for a spin-wait loop.

The duration of a PAUSE instruction's effect may vary significantly within and among implementations. In typical implementations this duration should be much less than the time to perform a context switch, probably more on the rough order of an on-chip cache miss latency or a cacheless access to main memory.

A series of PAUSE instructions can be used to create a cumulative delay loosely proportional to the number of PAUSE instructions. In spin-wait loops in portable code, however, only one PAUSE instruction should be used before re-evaluating loop conditions, else the hart might stall longer than optimal on some implementations, degrading system performance.

PAUSE is encoded as a FENCE instruction with *pred*=`W`, *succ*=`0`, *fm*=`0`, *rd*=`x0`, and *rs1*=`x0`.

PAUSE is encoded as a hint within the FENCE opcode because some implementations are expected to deliberately stall the PAUSE instruction until outstanding memory transactions have completed. Because the successor set is null, however, PAUSE does not mandate any particular memory ordering—hence, it truly is a HINT.

Like other FENCE instructions, PAUSE cannot be used within LR/SC sequences without voiding the forward-progress guarantee.

The choice of a predecessor set of W is arbitrary, since the successor set is null. Other HINTs similar to PAUSE might be encoded with other predecessor sets.

Chapter 10. "Zimop" Extension for May-Be-Operations, Version 1.0

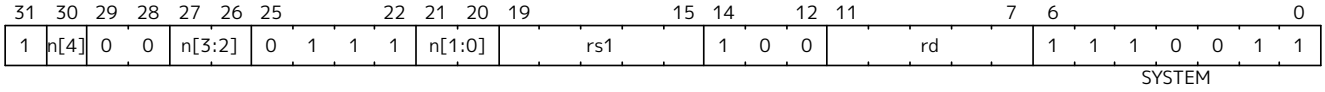
This chapter defines the "Zimop" extension, which introduces the concept of instructions that *may be operations* (MOPs). MOPs are initially defined to simply write zero to `x[rd]`, but are designed to be redefined by later extensions to perform some other action. The Zimop extension defines an encoding space for 40 MOPs.

It is sometimes desirable to define instruction-set extensions whose instructions, rather than raising illegal-instruction exceptions when the extension is not implemented, take no useful action (beyond writing `x[rd]`*). For example, programs with control-flow integrity checks can execute correctly on implementations without the corresponding extension, provided the checks are simply ignored. Implementing these checks as MOPs allows the same programs to run on implementations with or without the corresponding extension.*

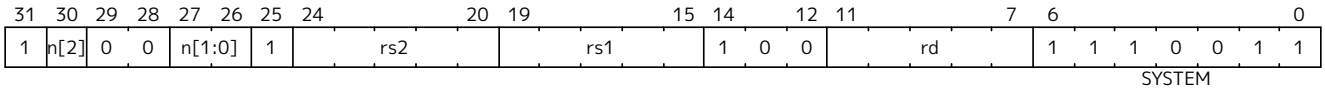
Although similar in some respects to HINTs, MOPs cannot be encoded as HINTs, because unlike HINTs, MOPs are allowed to alter architectural state.

Because MOPs may be redefined by later extensions, standard software should not execute a MOP unless it is deliberately targeting an extension that has redefined that MOP.

The Zimop extension defines 32 MOP instructions named MOP.R.*n*, where *n* is an integer between 0 and 31, inclusive. Unless redefined by another extension, these instructions simply write 0 to `x[rd]`. Their encoding allows future extensions to define them to read `x[rs1]`, as well as write `x[rd]`.



The Zimop extension additionally defines 8 MOP instructions named MOP.RR.*n*, where *n* is an integer between 0 and 7, inclusive. Unless redefined by another extension, these instructions simply write 0 to `x[rd]`. Their encoding allows future extensions to define them to read `x[rs1]` and `x[rs2]`, as well as write `x[rd]`.



The recommended assembly syntax for MOP.R.n is MOP.R.n rd, rs1, with any `x`*-register specifier being valid for either argument. Similarly for MOP.RR.n, the recommended syntax is MOP.RR.n rd, rs1, rs2. The extension that redefines a MOP may define an alternate assembly mnemonic.*

These MOPs are encoded in the SYSTEM major opcode in part because it is expected their behavior will be modulated by privileged CSR state.

These MOPs are defined to write zero to `x[rd]`*, rather than performing no operation, to simplify instruction decoding and to allow testing the presence of features by branching on the zeroness of the result.*

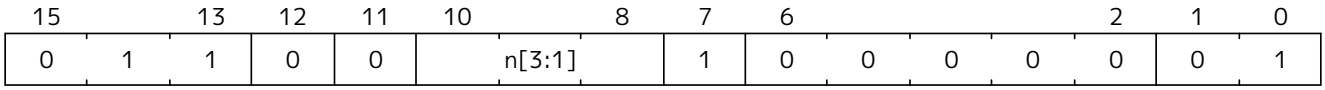
The MOPs defined in the Zimop extension do not carry a syntactic dependency from `x[rs1]` or `x[rs2]` to `x[rd]`, though an extension that redefines the MOP may impose such a requirement.

Not carrying a syntactic dependency relieves straightforward implementations of reading `x[rs1]` *and* `x[rs2]`.

10.1. "Zcmop" Compressed May-Be-Operations Extension, Version 1.0

This section defines the "Zcmop" extension, which defines eight 16-bit MOP instructions named C.MOP.*n*, where *n* is an odd integer between 1 and 15, inclusive. C.MOP.*n* is encoded in the reserved encoding space corresponding to C.LUI x*n*, 0, as shown in Table 10. Unlike the MOPs defined in the Zimop extension, the C.MOP.*n* instructions are defined to *not* write any register. Their encoding allows future extensions to define them to read register `x[n]`.

The Zcmop extension depends upon the Zca extension.



Very few suitable 16-bit encoding spaces exist. This space was chosen because it already has unusual behavior with respect to the `rd/rs1` *field—it encodes* `c.addi16sp` *when the field contains* `x2`*--and is therefore of lower value for most purposes.*

Table 10. C.MOP.n instruction encoding.

Mnemonic	Encoding	Redefinable to read register
C.MOP.1	`0110000010000001`	`x1`
C.MOP.3	`0110000110000001`	`x3`
C.MOP.5	`0110001010000001`	`x5`
C.MOP.7	`0110001110000001`	`x7`
C.MOP.9	`0110010010000001`	`x9`
C.MOP.11	`0110010110000001`	`x11`
C.MOP.13	`0110011010000001`	`x13`
C.MOP.15	`0110011110000001`	`x15`

The recommended assembly syntax for C.MOP.n is simply the nullary C.MOP.n. The possibly accessed register is implicitly `xn`*.*

The expectation is that each Zcmop instruction is equivalent to some Zimop instruction, but the choice of expansion (if any) is left to the extension that redefines the MOP. Note, a Zcmop instruction that does not write a value can expand into a write to `x0`*.*

Chapter 11. "Zicond" Extension for Integer Conditional Operations, Version 1.0.0

The Zicond extension defines two R-type instructions that support branchless conditional operations.

RV32	RV64	Mnemonic	Instruction
✓	✓	czero.eqz *rd*, *rs1*, *rs2*	Conditional zero, if condition is equal to zero
✓	✓	czero.nez *rd*, *rs1*, *rs2*	Conditional zero, if condition is nonzero

11.1. Instructions (in alphabetical order)

11.1.1. czero.eqz

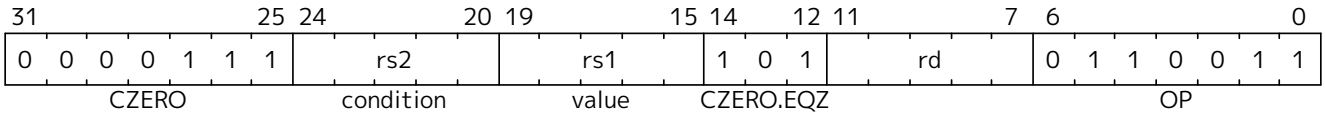
Synopsis

Moves zero to a register *rd*, if the condition *rs2* is equal to zero, otherwise moves *rs1* to *rd*.

Mnemonic

czero.eqz *rd*, *rs1*, *rs2*

Encoding



Description

If *rs2* contains the value zero, this instruction writes the value zero to *rd*. Otherwise, this instruction copies the contents of *rs1* to *rd*.

This instruction carries a syntactic dependency from both *rs1* and *rs2* to *rd*.

Furthermore, if the Zkt extension is implemented, this instruction's timing is independent of the data values in *rs1* and *rs2*.

SAIL code

```
let condition = X(rs2);
result : xlenbits = if (condition == zeros()) then zeros()
                                              else X(rs1);
X(rd) = result;
```

11.1.2. czero.nez

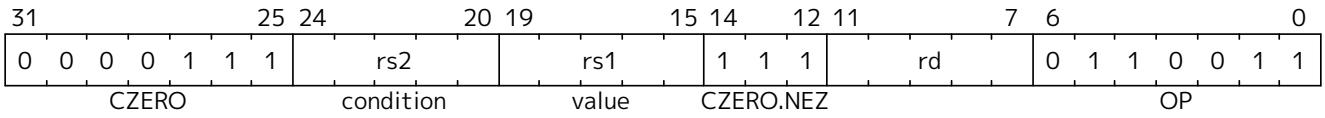
Synopsis

Moves zero to a register *rd*, if the condition *rs2* is nonzero, otherwise moves *rs1* to *rd*.

Mnemonic

czero.nez *rd*, *rs1*, *rs2*

Encoding



Description

If *rs2* contains a nonzero value, this instruction writes the value zero to *rd*. Otherwise, this instruction copies the contents of *rs1* to *rd*.

This instruction carries a syntactic dependency from both *rs1* and *rs2* to *rd*.

Furthermore, if the Zkt extension is implemented, this instruction's timing is independent of the data values in *rs1* and *rs2*.

SAIL code

```
let condition = X(rs2);
result : xlenbits = if (condition != zeros()) then zeros()
                                              else X(rs1);
X(rd) = result;
```

11.2. Usage examples

The instructions from this extension can be used to construct sequences that perform conditional-arithmetic, conditional-bitwise-logical, and conditional-select operations.

11.2.1. Instruction sequences

Operation	Instruction sequence	Length
Conditional add, if zero `rd = (rc == 0) ? (rs1 + rs2) : rs1`	`czero.nez  rd, rs2, rc` `add  rd, rs1, rd`	2 insns
Conditional add, if non-zero `rd = (rc != 0) ? (rs1 + rs2) : rs1`	`czero.eqz  rd, rs2, rc` `add  rd, rs1, rd`	
Conditional subtract, if zero `rd = (rc == 0) ? (rs1 - rs2) : rs1`	`czero.nez  rd, rs2, rc` `sub  rd, rs1, rd`	
Conditional subtract, if non-zero `rd = (rc != 0) ? (rs1 - rs2) : rs1`	`czero.eqz  rd, rs2, rc` `sub  rd, rs1, rd`	
Conditional bitwise-or, if zero `rd = (rc == 0) ? (rs1 | rs2) : rs1`	`czero.nez  rd, rs2, rc` `or  rd, rs1, rd`	
Conditional bitwise-or, if non-zero `rd = (rc != 0) ? (rs1 | rs2) : rs1`	`czero.eqz  rd, rs2, rc` `or  rd, rs1, rd`	
Conditional bitwise-xor, if zero `rd = (rc == 0) ? (rs1 ^ rs2) : rs1`	`czero.nez  rd, rs2, rc` `xor  rd, rs1, rd`	
Conditional bitwise-xor, if non-zero `rd = (rc != 0) ? (rs1 ^ rs2) : rs1`	`czero.eqz  rd, rs2, rc` `xor  rd, rs1, rd`	

Conditional bitwise-and, if zero
`rd = (rc == 0) ? (rs1 & rs2) : rs1`
	`and  rd, rs1, rs2`
	`czero.eqz  rtmp, rs1, rc`
	`or  rd, rd, rtmp`
		3 insns
		(requires 1 temporary)

Conditional bitwise-and, if non-zero
`rd = (rc != 0) ? (rs1 & rs2) : rs1`
	`and  rd, rs1, rs2`
	`czero.nez  rtmp, rs1, rc`
	`or  rd, rd, rtmp`	

Conditional select, if zero
`rd = (rc == 0) ? rs1 : rs2`
	`czero.nez  rd, rs1, rc`
	`czero.eqz  rtmp, rs2, rc`
	`add  rd, rd, rtmp`	

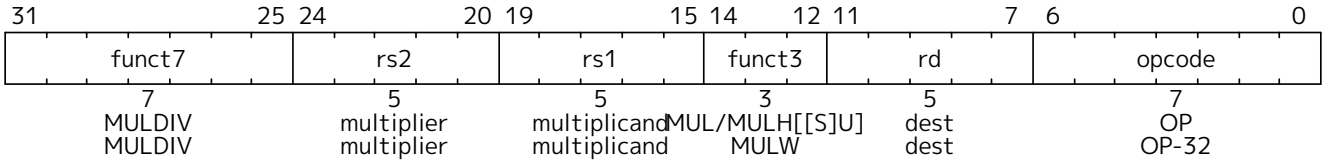
Conditional select, if non-zero
`rd = (rc != 0) ? rs1 : rs2`
	`czero.eqz  rd, rs1, rc`
	`czero.nez  rtmp, rs2, rc`
	`add  rd, rd, rtmp`	

Chapter 12. "M" Extension for Integer Multiplication and Division, Version 2.0

This chapter describes the standard integer multiplication and division instruction extension, which is named `M` and contains instructions that multiply or divide values held in two integer registers.

We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

12.1. Multiplication Operations



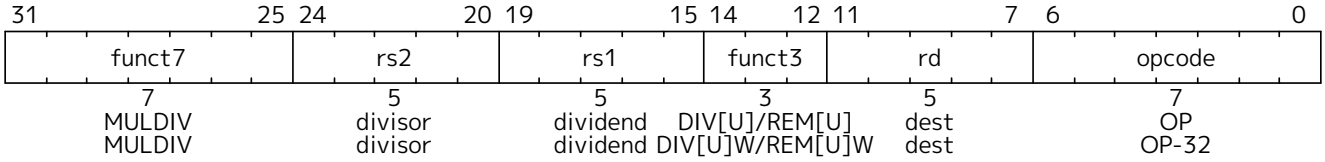
MUL performs an XLEN-bit×XLEN-bit multiplication of `rs1` by `rs2` and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and `rs1`×unsigned `rs2` multiplication. If both the high and low bits of the same product are required, then the recommended code sequence is: `MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2` (source register specifiers must be in same order and `rdh` cannot be the same as `rs1` or `rs2`). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the multiplicand (which contains the sign bit) with the less-significant words of the multiplier (which are unsigned).

MULW is an RV64 instruction that multiplies the lower 32 bits of the source registers, placing the sign extension of the lower 32 bits of the result into the destination register.

In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].

12.2. Division Operations



DIV and DIVU perform an XLEN bits by XLEN bits signed and unsigned integer division of `rs1` by `rs2`, rounding towards zero. REM and REMU provide the remainder of the corresponding division operation. For REM, the sign of a nonzero result equals the sign of the dividend.

For both signed and unsigned division, except in the case of overflow, it holds that dividend = divisor × quotient + remainder.

If both the quotient and remainder are required from the same division, the recommended code sequence is: `DIV[U] rdq, rs1, rs2; REM[U] rdr, rs1, rs2` (`rdq` cannot be the same as `rs1` or `rs2`). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

DIVW and DIVUW are RV64 instructions that divide the lower 32 bits of `rs1` by the lower 32 bits of `rs2`, treating them as signed and unsigned integers, placing the 32-bit quotient in `rd`, sign-extended to 64 bits. REMW and REMUW are RV64 instructions that provide the corresponding signed and unsigned remainder operations. Both REMW and REMUW always sign-extend the 32-bit result to 64 bits, including on a divide by zero.

The semantics for division by zero and division overflow are summarized in Table 11. The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by −1. The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

Table 11. Semantics for division by zero and division overflow. L is the width of the operation in bits: XLEN for DIV[U] and REM[U], or 32 for DIV[U]W and REM[U]W.

Condition	Dividend	Divisor	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	x	0	2^L-1	x	−1	x
Overflow (signed only)	-2^{L-1}	−1	-	-	-2^{L-1}	0

We considered raising exceptions on integer divide by zero, with these exceptions causing a trap in most execution environments. However, this would be the only arithmetic trap in the standard ISA (floating-point exceptions set flags and write default values, but do not cause traps) and would require language implementers to interact with the execution environment's trap handlers for this case. Further, where language standards mandate that a divide-by-zero exception must cause an immediate control flow change, only a single branch instruction needs to be added to each divide operation, and this branch instruction can be inserted after the divide and should normally be very predictably not taken, adding little runtime overhead.

The value of all bits set is returned for both unsigned and signed divide by zero to simplify the divider circuitry. The value of all 1s is both the natural value to return for unsigned divide, representing the largest unsigned number, and also the natural result for simple unsigned divider implementations. Signed division is often implemented using an unsigned division circuit and specifying the same overflow result simplifies the hardware.

12.3. `Zmmul` Extension, Version 1.0

The `Zmmul` extension implements the multiplication subset of the M extension. It adds all of the instructions defined in Section 12.1, namely: MUL, MULH, MULHU, MULHSU, and (for RV64 only) MULW. The encodings are identical to those of the corresponding M-extension instructions. `M` implies `Zmmul`.

The `Zmmul` extension enables low-cost implementations that require multiplication operations but not division. For many microcontroller applications, division operations are too infrequent to justify the cost of divider hardware. By contrast, multiplication operations are more frequent, making the cost of multiplier hardware more justifiable. Simple FPGA soft cores particularly benefit from eliminating division but retaining multiplication, since many FPGAs provide hardwired multipliers but require dividers be implemented in soft logic.

Chapter 13. "A" Extension for Atomic Instructions, Version 2.1

The atomic-instruction extension, named "A", contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model. (Gharachorloo et al., 1990)

After much debate, the language community and architecture community appear to have finally settled on release consistency as the standard memory consistency model and so the RISC-V atomic support is built around this model.

The A extension comprises instructions provided by the Zaamo and Zalrsc extensions.

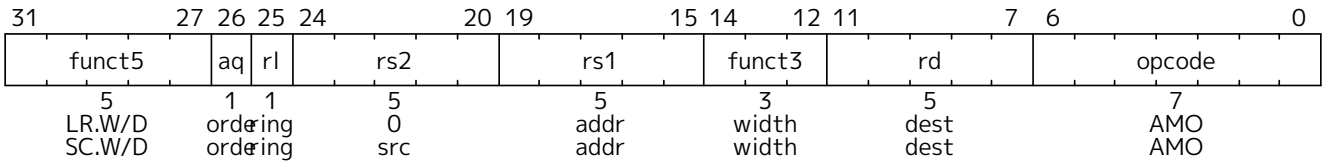
13.1. Specifying Ordering of Atomic Instructions

The base RISC-V ISA has a relaxed memory model, with the FENCE instruction used to impose additional ordering constraints. The address space is divided by the execution environment into memory and I/O domains, and the FENCE instruction provides options to order accesses to one or both of these two address domains.

To provide more efficient support for release consistency (Gharachorloo et al., 1990), each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V harts. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains.

If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the *aq* bit is set, the atomic memory operation is treated as an *acquire* access, i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation. If only the *rl* bit is set, the atomic memory operation is treated as a *release* access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the *aq* and *rl* bits are set, the atomic memory operation is *sequentially consistent* and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

13.2. "Zalrsc" Extension for Load-Reserved/Store-Conditional Instructions



Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR.W loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a *reservation set*—a set of bytes that subsumes the bytes in the addressed word. SC.W conditionally writes a word in *rs2* to the address in *rs1*: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in *rs2* to memory, and it writes zero to *rd*. If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to *rd*. No SC.W instruction shall retire unless it passes

memory permission checks, but it is UNSPECIFIED whether any side effects of implicit address translation and protection memory accesses (such as setting a page-table entry D bit) occur on a failed SC.W. For the purposes of memory protection, a failed SC.W may be treated like a store. Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart. LR.D and SC.D act analogously on doublewords and are only available on RV64. For RV64, LR.W and SC.W sign-extend the value placed in *rd*.

Both compare-and-swap (CAS) and LR/SC can be used to build lock-free data structures. After extensive discussion, we opted for LR/SC for several reasons: 1) CAS suffers from the ABA problem, which LR/SC avoids because it monitors all writes to the address rather than only checking for changes in the data value; 2) CAS would also require a new integer instruction format to support three source operands (address, compare value, swap value) as well as a different memory system message format, which would complicate microarchitectures; 3) Furthermore, to avoid the ABA problem, other systems provide a double-wide CAS (DW-CAS) to allow a counter to be tested and incremented along with a data word. This requires reading five registers and writing two in one instruction, and also a new larger memory system message type, further complicating implementations; 4) LR/SC provides a more efficient implementation of many primitives as it only requires one load as opposed to two with CAS (one load before the CAS instruction to obtain a value for speculative computation, then a second load as part of the CAS instruction to check if value is unchanged before updating).

The main disadvantage of LR/SC over CAS is livelock, which we avoid, under certain circumstances, with an architected guarantee of eventual forward progress as described below. Another concern is whether the influence of the current x86 architecture, with its DW-CAS, will complicate porting of synchronization libraries and other software that assumes DW-CAS is the basic machine primitive. A possible mitigating factor is the recent addition of transactional memory instructions to x86, which might cause a move away from DW-CAS.

More generally, a multi-word atomic primitive is desirable, but there is still considerable debate about what form this should take, and guaranteeing forward progress adds complexity to a system.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

We reserve a failure code of 1 to mean "unspecified" so that simple implementations may return this value using the existing multiplexer required for the SLT/SLTU instructions. More specific failure codes might be defined in future versions or extensions to the ISA.

For LR and SC, the Zalrsc extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for *doublewords* and four-byte aligned for *words*). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself

in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom in Section 18.1.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- *during a preemptive context switch, and*
- *if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C++ `memory_order_seq_cst` *sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C++ atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.*

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set, nor should software set the *aq* bit on an SC instruction unless the *rl* bit is also set. LR.*rl* and SC.*aq* instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

```
        # a0 holds address of memory location
        # a1 holds expected value
        # a2 holds desired value
        # a0 holds return value, 0 if successful, !0 otherwise
    cas:
        lr.w t0, (a0)         # Load original value.
        bne t0, a1, fail      # Doesn't match, so fail.
        sc.w t0, a2, (a0)     # Try to update.
        bnez t0, cas          # Retry if store-conditional failed.
        li a0, 0              # Set return to success.
        jr ra                 # Return.
    fail:
        li a0, 1              # Set return to failure.
        jr ra                 # Return.
```

Listing 2. Sample code for compare-and-swap function using LR/SC.

LR/SC can be used to construct lock-free data structures. An example using LR/SC to implement a compare-and-swap function is shown in Listing 2. If inlined, compare-and-swap functionality need only take four instructions.

13.3. Eventual Success of Store-Conditional Instructions

The Zalrsc extension defines *constrained LR/SC loops*, which have the following properties:

- The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory.
- An LR/SC sequence begins with an LR instruction and ends with an SC instruction. The dynamic code executed between the LR and SC instructions can only contain instructions from the base "I" instruction set, excluding loads, stores, backward jumps, taken backward branches, JALR, FENCE, and SYSTEM instructions. Compressed forms of the aforementioned "I" instructions in the C (hence Zca) and Zcb extensions are also permitted.
- The code to retry a failing LR/SC sequence can contain backwards jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraint as the code between the LR and SC.
- The LR and SC addresses must lie within a memory region with the *LR/SC eventuality* property. The execution environment is responsible for communicating which regions have this property.
- The SC must be to the same effective address and of the same data size as the latest LR executed by the same hart.

LR/SC sequences that do not lie within constrained LR/SC loops are *unconstrained*. Unconstrained LR/SC sequences might succeed on some attempts on some implementations, but might never succeed on other implementations.

We restricted the length of LR/SC loops to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and associativity. Similarly, we disallowed other loads and stores within the loops to avoid restrictions on data-

cache associativity in simple implementations that track the reservation within a private cache. The restrictions on branches and jumps limit the time that can be spent in the sequence. Floating-point operations and integer multiply/divide were disallowed to simplify the operating system's emulation of these instructions on implementations lacking appropriate hardware support.

Software is not forbidden from using unconstrained LR/SC sequences, but portable software must detect the case that the sequence repeatedly fails, then fall back to an alternate code sequence that does not rely on an unconstrained LR/SC sequence. Implementations are permitted to unconditionally fail any unconstrained LR/SC sequence.

[#norm:constrained_lrsc_forward_progress]#If a hart *H* enters a constrained LR/SC loop, the execution environment must guarantee that one of the following events eventually occurs:

- *H* or some other hart executes a successful SC to the reservation set of the LR instruction in *H*'s constrained LR/SC loops.
- Some other hart executes an unconditional store or AMO instruction to the reservation set of the LR instruction in *H*'s constrained LR/SC loop, or some other device in the system writes to that reservation set.
- *H* executes a branch or jump that exits the constrained LR/SC loop.
- *H* traps.#

Note that these definitions permit an implementation to fail an SC instruction occasionally for any reason, provided the aforementioned guarantee is not violated.

As a consequence of the eventuality guarantee, if some harts in an execution environment are executing constrained LR/SC loops, and no other harts or devices in the execution environment execute an unconditional store or AMO to that reservation set, then at least one hart will eventually exit its constrained LR/SC loop. By contrast, if other harts or devices continue to write to that reservation set, it is not guaranteed that any hart will exit its LR/SC loop.

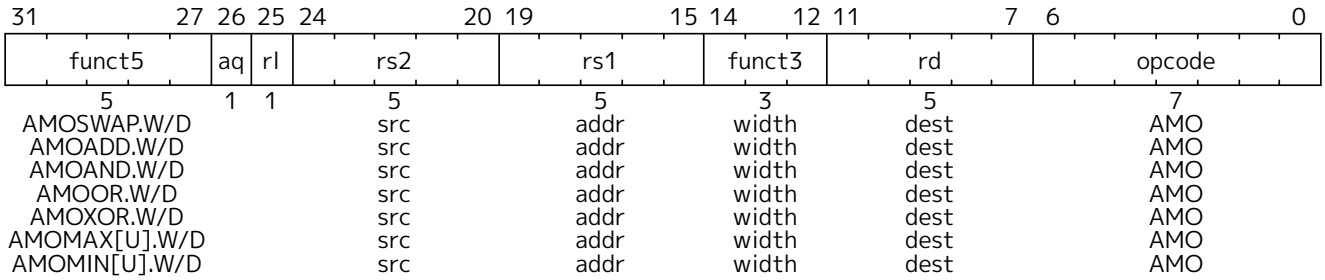
Loads and load-reserved instructions do not by themselves impede the progress of other harts' LR/SC sequences. We note this constraint implies, among other things, that loads and load-reserved instructions executed by other harts (possibly within the same core) cannot impede LR/SC progress indefinitely. For example, cache evictions caused by another hart sharing the cache cannot impede LR/SC progress indefinitely. Typically, this implies reservations are tracked independently of evictions from any shared cache. Similarly, cache misses caused by speculative execution within a hart cannot impede LR/SC progress indefinitely.

These definitions admit the possibility that SC instructions may spuriously fail for implementation reasons, provided progress is eventually made.

One advantage of CAS is that it guarantees that some hart eventually makes progress, whereas an LR/SC atomic sequence could livelock indefinitely on some systems. To avoid this concern, we added an architectural guarantee of livelock freedom for certain LR/SC sequences.

Earlier versions of this specification imposed a stronger starvation-freedom guarantee. However, the weaker livelock-freedom guarantee is sufficient to implement the C11 and C++11 languages, and is substantially easier to provide in some microarchitectural styles.

13.4. "Zaamo" Extension for Atomic Memory Operations



The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO instructions atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a binary operator to the loaded value and the original value in *rs2*, then store the result back to the original address in *rs1*. AMOs can either operate on *doublewords* (RV64 only) or *words* in memory. For RV64, 32-bit AMOs always sign-extend the value placed in *rd*, and ignore the upper 32 bits of the original value of *rs2*.

For AMOs, the Zaamo extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for *doublewords* and four-byte aligned for *words*). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

The misaligned atomicity granule PMA, defined in Volume II of this manual, optionally relaxes this alignment requirement. If present, the misaligned atomicity granule PMA specifies the size of a misaligned atomicity granule, a power-of-two number of bytes. The misaligned atomicity granule PMA applies only to AMOs, loads and stores defined in the base ISAs, and loads and stores of no more than XLEN bits defined in the F, D, and Q extensions, and compressed encodings thereof. For an instruction in that set, if all accessed bytes lie within the same misaligned atomicity granule, the instruction will not raise an exception for reasons of address alignment, and the instruction will give rise to only one memory operation for the purposes of RVWMO—i.e., it will execute atomically.

The operations supported are swap, integer add, bitwise AND, bitwise OR, bitwise XOR, and signed and unsigned integer maximum and minimum. Without ordering constraints, these AMOs can be used to implement parallel reduction operations, where typically the return value would be discarded by writing to `x0`.

We provided fetch-and-op style atomic primitives as they scale to highly parallel systems better than LR/SC or CAS. A simple microarchitecture can implement AMOs using the LR/SC primitives, provided the implementation can guarantee the AMO eventually completes. More complex implementations might also implement AMOs at memory controllers, and can optimize away fetching the original value when the destination is `x0`.

The set of AMOs was chosen to support the C11/C++11 atomic memory operations efficiently, and also to support parallel reductions in memory. Another use of AMOs is to provide atomic updates to memory-mapped device registers (e.g., setting, clearing, or toggling bits) in the I/O space.

The Zaamo extension enables microcontroller class implementations to utilize atomic primitives from the AMO subset of the A extension. Typically such implementations do not have caches and thus may not be able to naturally support the LR/SC instructions provided by the Zalrsc extension.

To help implement multiprocessor synchronization, the AMOs optionally provide release consistency

semantics. If the *aq* bit is set, then no later memory operations in this RISC-V hart can be observed to take place before the AMO. Conversely, if the *rl* bit is set, then other RISC-V harts will not observe the AMO before memory accesses preceding the AMO in this RISC-V hart. Setting both the *aq* and the *rl* bit on an AMO makes the sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

The AMOs were designed to implement the C11 and C++11 memory models efficiently. Although the FENCE R, RW instruction suffices to implement the acquire operation and FENCE RW, W suffices to implement release, both imply additional unnecessary ordering as compared to AMOs with the corresponding aq or rl bit set.

An example code sequence for a critical section guarded by a test-and-test-and-set spinlock is shown in Example Listing 3. Note the first AMO is marked aq to order the lock acquisition before the critical section, and the second AMO is marked rl to order the critical section before the lock relinquishment.

```
    li           t0, 1        # Initialize swap value.
again:
    lw           t1, (a0)     # Check if lock is held.
    bnez         t1, again    # Retry if held.
    amoswap.w.aq t1, t0, (a0) # Attempt to acquire lock.
    bnez         t1, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Listing 3. Sample code for mutual exclusion. `a0` *contains the address of the lock.*

We recommend the use of the AMO Swap idiom shown in Listing 3 for both lock acquire and release to simplify the implementation of speculative lock elision. (Rajwar & Goodman, 2001)

The instructions in the "A" extension can be used to provide sequentially consistent loads and stores, but this constrains hardware reordering of memory accesses more than necessary. A C++ sequentially consistent load can be implemented as an LR with aq set. However, the LR/SC eventual success guarantee may slow down concurrent loads from the same effective address. A sequentially consistent store can be implemented as an AMOSWAP that writes the old value to `x0` *and has rl set. However the superfluous load may impose ordering constraints that are unnecessary for this use case. Specific compilation conventions may require both the aq and rl bits to be set in either or both the LR and AMOSWAP instructions.*

Chapter 14. "Zawrs" Extension for Wait-on-Reservation-Set instructions, Version 1.01

The Zawrs extension defines a pair of instructions to be used in polling loops that allows a core to enter a low-power state and wait on a store to a memory location. Waiting for a memory location to be updated is a common pattern in many use cases such as:

1. Contenders for a lock waiting for the lock variable to be updated.
2. Consumers waiting on the tail of an empty queue for the producer to queue work/data. The producer may be code executing on a RISC-V hart, an accelerator device, an external I/O agent.
3. Code waiting on a flag to be set in memory indicative of an event occurring. For example, software on a RISC-V hart may wait on a "done" flag to be set in memory by an accelerator device indicating completion of a job previously submitted to the device.

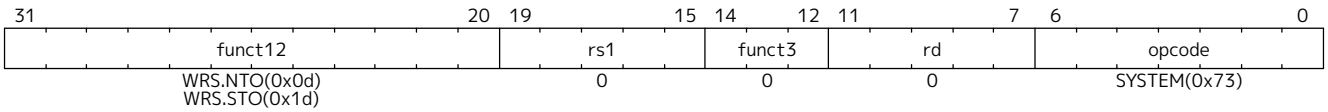
Such use cases involve polling on memory locations, and such busy loops can be a wasteful expenditure of energy. To mitigate the wasteful looping in such usages, a `WRS.NTO` (WRS-with-no-timeout) instruction is provided. Instead of polling for a store to a specific memory location, software registers a reservation set that includes all the bytes of the memory location using the `LR` instruction. Then a subsequent `WRS.NTO` instruction would cause the hart to temporarily stall execution in a low-power state until a store occurs to the reservation set or an interrupt is observed.

Sometimes the program waiting on a memory update may also need to carry out a task at a future time or otherwise place an upper bound on the wait. To support such use cases a second instruction `WRS.STO` (WRS-with-short-timeout) is provided that works like `WRS.NTO` but bounds the stall duration to an implementation-define short timeout such that the stall is terminated on the timeout if no other conditions have occurred to terminate the stall. The program using this instruction may then determine if its deadline has been reached.

The instructions in the Zawrs extension are only useful in conjunction with the LR instruction, which is provided by the Zalrsc component of the A extension.

14.1. Wait-on-Reservation-Set Instructions

The `WRS.NTO` and `WRS.STO` instructions cause the hart to temporarily stall execution in a low-power state as long as the reservation set is valid and no pending interrupts, even if disabled, are observed. For `WRS.STO` the stall duration is bounded by an implementation defined short timeout. These instructions are available in all privilege modes.



Hart execution may be stalled while the following conditions are all satisfied:

a. The reservation set is valid

b. If `WRS.STO`, a "short" duration since start of stall has not elapsed

c. No pending interrupt is observed (see the rules below)

While stalled, an implementation is permitted to occasionally terminate the stall and complete execution for any reason.

`WRS.NTO` and `WRS.STO` instructions follow the rules of the `WFI` instruction for resuming execution on a pending interrupt.

When the `TW` (Timeout Wait) bit in `mstatus` is set and `WRS.NTO` is executed in any privilege mode other than M mode, and it does not complete within an implementation-specific bounded time limit, the `WRS.NTO` instruction will cause an illegal-instruction exception.

When executing in VS or VU mode, if the `VTW` bit is set in `hstatus`, the `TW` bit in `mstatus` is clear, and the `WRS.NTO` does not complete within an implementation-specific bounded time limit, the `WRS.NTO` instruction will cause a virtual-instruction exception.

Since the `WRS.STO` *and* `WRS.NTO` *instructions can complete execution for reasons other than stores to the reservation set, software will likely need a means of looping until the required stores have occurred.*

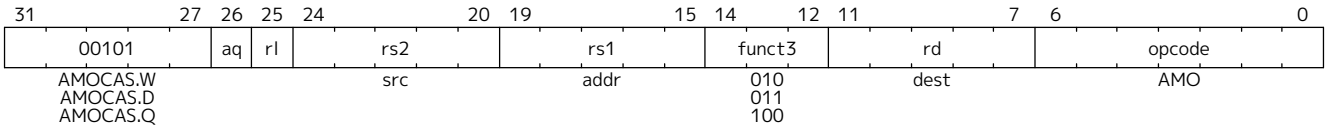
The duration of a `WRS.STO` *instruction's timeout may vary significantly within and among implementations. In typical implementations this duration should be roughly in the range of 10 to 100 times an on-chip cache miss latency or a cacheless access to main memory.*

`WRS.NTO`*, unlike* `WFI`*, is not specified to cause an illegal-instruction exception if executed in U-mode when the governing* `TW` *bit is 0.* `WFI` *is typically not expected to be used in U-mode and on many systems may promptly cause an illegal-instruction exception if used at U-mode. Unlike* `WFI`*,* `WRS.NTO` *is expected to be used by software in U-mode when waiting on memory but without a deadline for that wait.*

Chapter 15. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions, Version 1.0.0

Compare-and-Swap (CAS) provides an easy and typically faster way to perform thread synchronization operations when supported as a hardware instruction. CAS is typically used by lock-free and wait-free algorithms. This extension defines CAS instructions to operate on 32-bit, 64-bit, and 128-bit (RV64 only) data values. The Zacas extension depends upon the Zaamo extension.

15.1. Word/Doubleword/Quadword CAS (AMOCAS.W/D/Q) Instructions



For RV32, `AMOCAS.W` atomically loads a 32-bit data value from address in `rs1`, compares the loaded value to the 32-bit value held in `rd`, and if the comparison is bitwise equal, then stores the 32-bit value held in `rs2` to the original address in `rs1`. The value loaded from memory is placed into register `rd`. The operation performed by `AMOCAS.W` for RV32 is as follows:

```
temp = mem[X(rs1)]
if ( temp == X(rd) )
    mem[X(rs1)] = X(rs2)
X(rd) = temp
```

`AMOCAS.D` is similar to `AMOCAS.W` but operates on 64-bit data values.

For RV32, `AMOCAS.D` atomically loads 64-bits of a data value from address in `rs1`, compares the loaded value to a 64-bit value held in a register pair consisting of `rd` and `rd+1`, and if the comparison is bitwise equal, then stores the 64-bit value held in the register pair `rs2` and `rs2+1` to the original address in `rs1`. The value loaded from memory is placed into the register pair `rd` and `rd+1`. The instruction requires the first register in the pair to be even numbered; encodings with odd numbered registers specified in `rs2` and `rd` are reserved. When the first register of a source register pair is `x0`, then both halves of the pair read as zero. When the first register of a destination register pair is `x0`, then the entire register result is discarded and neither destination register is written. The operation performed by `AMOCAS.D` for RV32 is as follows:

```
temp0 = mem[X(rs1)+0]
temp1 = mem[X(rs1)+4]
comp0 = (rd == x0)  ? 0 : X(rd)
comp1 = (rd == x0)  ? 0 : X(rd+1)
swap0 = (rs2 == x0) ? 0 : X(rs2)
swap1 = (rs2 == x0) ? 0 : X(rs2+1)
if ( temp0 == comp0 ) && ( temp1 == comp1 )
    mem[X(rs1)+0] = swap0
    mem[X(rs1)+4] = swap1
endif
if ( rd != x0 )
    X(rd)   = temp0
    X(rd+1) = temp1
endif
```

For RV64, `AMOCAS.W` atomically loads a 32-bit data value from address in `rs1`, compares the loaded value to the lower 32 bits of the value held in `rd`, and if the comparison is bitwise equal, then stores the lower 32 bits of the value held in `rs2` to the original address in `rs1`. The 32-bit value loaded from memory is sign-extended and is placed into register `rd`. The operation performed by `AMOCAS.W` for RV64 is as follows:

```
temp[31:0] = mem[X(rs1)]
if ( temp[31:0] == X(rd)[31:0] )
    mem[X(rs1)] = X(rs2)[31:0]
X(rd) = SignExtend(temp[31:0])
```

For RV64, `AMOCAS.D` atomically loads 64-bits of a data value from address in `rs1`, compares the loaded value to a 64-bit value held in `rd`, and if the comparison is bitwise equal, then stores the 64-bit value held in `rs2` to the original address in `rs1`. The value loaded from memory is placed into register `rd`. The operation performed by `AMOCAS.D` for RV64 is as follows:

```
temp = mem[X(rs1)]
if ( temp == X(rd) )
    mem[X(rs1)] = X(rs2)
X(rd) = temp
```

`AMOCAS.Q` (RV64 only) atomically loads 128-bits of a data value from address in `rs1`, compares the loaded value to a 128-bit value held in a register pair consisting of `rd` and `rd+1`, and if the comparison is bitwise equal, then stores the 128-bit value held in the register pair `rs2` and `rs2+1` to the original address in `rs1`. The value loaded from memory is placed into the register pair `rd` and `rd+1`. The instruction requires the first register in the pair to be even numbered; encodings with odd numbered registers specified in `rs2` and `rd` are reserved. When the first register of a source register pair is `x0`, then both halves of the pair read as zero. When the first register of a destination register pair is `x0`, then the entire register result is discarded and neither destination register is written. The operation performed by `AMOCAS.Q` is as follows:

```
temp0 = mem[X(rs1)+0]
temp1 = mem[X(rs1)+8]
comp0 = (rd == x0)  ? 0 : X(rd)
comp1 = (rd == x0)  ? 0 : X(rd+1)
swap0 = (rs2 == x0) ? 0 : X(rs2)
swap1 = (rs2 == x0) ? 0 : X(rs2+1)
if ( temp0 == comp0 ) && ( temp1 == comp1 )
    mem[X(rs1)+0] = swap0
    mem[X(rs1)+8] = swap1
endif
if ( rd != x0 )
    X(rd)   = temp0
    X(rd+1) = temp1
endif
```

Some algorithms may load the previous data value of a memory location into the register used as the compare data value source by a Zacas instruction. When using a Zacas instruction that uses a register pair to source the compare value, the two registers may be loaded using two individual loads. The two individual loads may read an inconsistent pair of values but that is not an issue since the `AMOCAS` *operation itself uses an atomic load-pair from memory to obtain the data value for its comparison.*

The following example code sequence illustrates the use of `AMOCAS.D` *in a RV32 implementation to atomically increment a 64-bit counter.*

```
# a0 - address of the counter.
increment:
  lw   a2, (a0)      # Load current counter value using
  lw   a3, 4(a0)     # two individual loads.
retry:
  mv   a6, a2        # Save the low 32 bits of the current value.
  mv   a7, a3        # Save the high 32 bits of the current
value.
  addi a4, a2, 1     # Increment the low 32 bits.
  sltu a1, a4, a2    # Determine if there is a carry out.
  add  a5, a3, a1    # Add the carry if any to high 32 bits.
  amocas.d.aqrl a2, a4, (a0)
  bne  a2, a6, retry # If amocas.d failed then retry
  bne  a3, a7, retry # using current values loaded by amocas.d.
  ret
```

Just as for AMOs in the A extension, `AMOCAS.W/D/Q` requires that the address held in `rs1` be naturally aligned to the size of the operand (i.e., 16-byte aligned for *quadwords*, eight-byte aligned for *doublewords*, and four-byte aligned for *words*). And the same exception options apply if the address is not naturally aligned.

Just as for AMOs in the A extension, the `AMOCAS.W/D/Q` optionally provide release consistency semantics, using the `aq` and `rl` bits, to help implement multiprocessor synchronization. The memory operation performed by an `AMOCAS.W/D/Q`, when successful, has acquire semantics if `aq` bit is 1 and has release

semantics if `rl` bit is 1. The memory operation performed by an `AMOCAS.W/D/Q`, when not successful, has acquire semantics if `aq` bit is 1 but does not have release semantics, regardless of `rl`.

A FENCE instruction may be used to order the memory read access and, if produced, the memory write access by an `AMOCAS.W/D/Q` instruction.

An unsuccessful `AMOCAS.W/D/Q` *may either not perform a memory write or may write back the old value loaded from memory. The memory write, if produced, does not have release semantics, regardless of* `rl`*. Irrespective of whether a write is actually performed, the instruction is treated as an AMO for the purposes of the RVWMO PPO rules.*

An `AMOCAS.W/D/Q` instruction always requires write permissions.

The following example code sequence illustrates the use of `AMOCAS.Q` *to implement the enqueue operation for a non-blocking concurrent queue using the algorithm outlined in (Michael & Scott, 1996). The algorithm atomically operates on a pointer and its associated modification counter using the* `AMOCAS.Q` *instruction to avoid the ABA problem.*

```
# Enqueue operation of a non-blocking concurrent queue.
# Data structures used by the queue:
#   structure pointer_t {ptr:   node_t *, count: uint64_t}
#   structure node_t    {next: pointer_t, value: data type}
#   structure queue_t   {Head: pointer_t, Tail:  pointer_t}
# Inputs to the procedure:
#   a0 - address of Tail variable
#   a4 - address of a new node to insert at tail
enqueue:
  ld   a6, (a0)          # a6 = Tail.ptr
  ld   a7, 8(a0)         # a7 = Tail.count
  ld   a2, (a6)          # a2 = Tail.ptr->next.ptr
  ld   a3, 8(a6)         # a3 = Tail.ptr->next.count
  ld   t1, (a0)
  ld   t2, 8(a0)
  bne  a6, t1, enqueue   # Retry if Tail & next are not
consistent
  bne  a7, t2, enqueue   # Retry if Tail & next are not
consistent
  bne  a2, x0, move_tail # Was tail pointing to the last node?
  mv   t1, a2            # Save Tail.ptr->next.ptr
  mv   t2, a3            # Save Tail.ptr->next.count
  addi a5, a3, 1         # Link the node at the end of the list
  amocas.q.aqrl a2, a4, (a6)
  bne  a2, t1, enqueue   # Retry if CAS failed
  bne  a3, t2, enqueue   # Retry if CAS failed
  addi a5, a7, 1         # Update Tail to the inserted node
  amocas.q.aqrl a6, a4, (a0)
  ret                    # Enqueue done
move_tail:               # Tail was not pointing to the last node
  addi a3, a7, 1         # Try to swing Tail to the next node
  amocas.q.aqrl a6, a2, (a0)
```

```
    j     enqueue           # Retry
```

Chapter 16. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0

The A-extension offers atomic memory operation (AMO) instructions for *words*, *doublewords*, and *quadwords* (only for `AMOCAS`). The absence of atomic operations for subword data types necessitates emulation strategies. For bitwise operations, this emulation can be performed via word-sized bitwise AMO* instructions. For non-bitwise operations, emulation is achievable using word-sized `LR`/`SC` instructions.

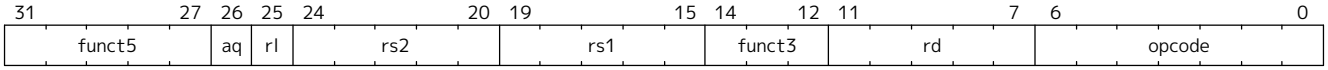
Several limitations arise from this emulation approach:

1. In systems with large-scale or Non-Uniform Memory Access (NUMA) configurations, emulation based on `LR`/`SC` introduces issues related to scalability and fairness, particularly under conditions of high contention.
2. Emulation of narrower AMOs through wider AMO* instructions on non-idempotent IO memory regions may result in unintended side effects.
3. Utilizing wider AMO* instructions for emulating narrower AMOs risks activating extraneous breakpoints or watchpoints.
4. In the absence of native support for subword atomics, compilers often resort to inlining code sequences to provide the required emulation. This practice contributes to an increase in code size, with consequent impacts on system performance and memory utilization.

The Zabha extension addresses these limitations by adding support for *byte* and *halfword* atomic memory operations to the RISC-V Unprivileged ISA. The Zabha extension depends upon the Zaamo standard extension.

16.1. Byte and Halfword Atomic Memory Operation Instructions

Zabha extension provides the `AMO[ADD|AND|OR|XOR|SWAP|MIN[U]|MAX[U]].[B|H]` instructions. If Zacas extension is also implemented, Zabha further provides the `AMOCAS.[B|H]` instructions.



AMOSWAP.B/H	ordering	src	addr	width=0/1	dest	AMO
AMOADD.B/H	ordering	src	addr	width=0/1	dest	AMO
AMOAND.B/H	ordering	src	addr	width=0/1	dest	AMO
AMOOR.B/H	ordering	src	addr	width=0/1	dest	AMO
AMOXOR.B/H	ordering	src	addr	width=0/1	dest	AMO
AMOMAX[U].B/H	ordering	src	addr	width=0/1	dest	AMO
AMOMIN[U].B/H	ordering	src	addr	width=0/1	dest	AMO
AMOCAS.B/H	ordering	src	addr	width=0/1	dest	AMO

Byte and halfword AMOs always sign-extend the value placed in `rd`, and ignore the $XLEN-1:2^{(width+3)}$ bits of the original value in `rs2`. The `AMOCAS.[B|H]` instructions similarly ignore the $XLEN-1:2^{(width+3)}$ bits of the original value in `rd`.

Similar to the AMOs specified in the A extension, the Zabha extension mandates that the address contained in the `rs1` register must be naturally aligned to the size of the operand. The same exception options as specified in the A extension are applicable in cases where the address is not naturally aligned.

Similar to the AMOs specified in the A and Zacas extensions, the AMOs in the Zabha extension optionally provide release consistency semantics, using the `aq` and `rl` bits, to help implement multiprocessor synchronization.

Zabha omits byte and halfword support for `LR` and `SC` due to low utility.

Chapter 17. "Zalasr" Atomic Load-Acquire and Store-Release Instructions, Version 1.0

The Zalasr (Load-Acquire and Store-Release) extension provides load-acquire and store-release instructions in RISC-V. These can be important for high performance designs by enabling finer-grained synchronisation than is possible with fences alone, by providing a unidirectional fence. Load-acquire and store-release are widely used in language-level memory models: both the Java and C++ memory models make use of acquire-release semantics, and C++'s `atomic` provides primitives that are meant to map directly to load-acquire and store-release instructions.

The Zalasr extension builds on the atomic support provided by the Zaamo (Atomic Memory Operations), Zalrsc (Load-Reserved and Store-Conditional), and Zabha (Byte and Halfword Atomic Memory Operations) extensions by providing additional atomic operations (although it can be implemented independently of them). All of the AMO operations in Zaamo (and Zabha) are read-modify-write operations that both load and store. The Zalrsc extension provides operations that are only loads or stores. However, since it is designed to perform an atomic operation on a single memory word or doubleword, the loads and stores are designed to be paired. The load-reserved implies that a future store-conditional will follow while store-conditional requires that there was a previous load-reserved without other intervening loads or stores. Therefore, the Zalrsc extension does not provide a general atomic and ordered load or store.

Zalasr fills this gap by offering truly standalone atomic and ordered loads and stores. The Zalasr instructions are atomic loads and stores that support ordering annotations. With the combination of Zaamo, Zabha, and Zalasr all C++ atomic operations can be supported with single instructions.

17.1. Load-Acquire and Store-Release Instructions

The Zalasr instructions always sign-extend the value placed in *rd* and ignore the upper bits of the value of *rs2*. The instructions in the Zalasr extension require that the address held in *rs1* be naturally aligned to the size in bytes (2^{width}) of the operand. If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

The misaligned atomicity granule PMA, defined in Volume II of this manual, optionally relaxes this alignment requirement. If all accessed bytes lie within the same misaligned atomicity granule, the instruction will not raise an exception for reasons of address alignment, and the instruction will give rise to only one memory operation for the purposes of RVWMO—i.e., it will execute atomically.

17.2. Load Acquire

Synopsis

The load-acquire instruction atomically loads a 2^{width}-byte value from the address in *rs1* and places the sign-extended value into the register *rd*, subject to the ordering annotations specified in the instruction.

Mnemonic

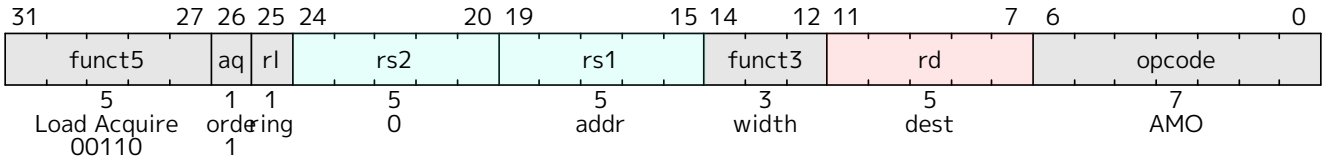
lb.{aq,aqrl} *rd*, (*rs1*)

lh.{aq,aqrl} *rd*, (*rs1*)

lw.{aq,aqrl} *rd*, (*rs1*)

ld.{aq,aqrl} *rd*, (*rs1*)

Encoding



Description

This instruction loads 2^{width} bytes of memory from rs1 atomically and writes the result into rd. If the size ($2^{width+3}$) is less than XLEN, it is sign-extended to fill the destination register. This load must have the ordering annotation *aq* and may have ordering annotation *rl* encoded in the instruction. The instruction always has an "acquire-RCsc" annotation, and if the bit *rl* is set the instruction has a "release-RCsc" annotation.

The versions without the *aq* bit set are RESERVED. LD.{AQ, AQRL} is RV64-only.

The aq bit is mandatory because the two encodings that would be produced are not seen as useful at this time. The version with neither the aq nor the rl bit set would correspond to a load with no ordering annotations that was guaranteed to be performed atomically. This can be achieved with ordinary load instructions by suitably aligning pointers. The version with only the rl bit would correspond to load-release. Load-release has theoretical applications in seqlocks, but is not supported in language-level memory models and so is not included.

17.3. Store Release

Synopsis

The store-release instruction atomically stores the 2^{width}-byte value from the low bits of register *rs2* to the address in *rs1*, subject to the ordering annotations specified in the instruction.

Mnemonic

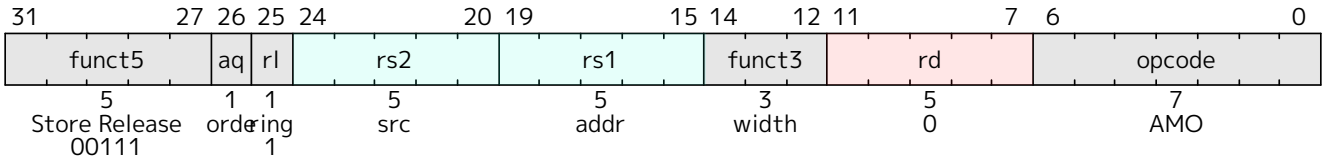
sb.{rl,aqrl} *rs2*, (*rs1*)

sh.{rl,aqrl} *rs2*, (*rs1*)

sw.{rl,aqrl} *rs2*, (*rs1*)

sd.{rl,aqrl} *rs2*, (*rs1*)

Encoding



Description

This instruction stores 2^{width} bytes of memory from rs1 atomically. This store must have ordering annotation *rl* and may have ordering annotation *aq* encoded in the instruction. The instruction always has an "release-RCsc" annotation, and if the bit *aq* is set the instruction has a "acquire-RCsc" annotation.

The versions without the *rl* bit set are RESERVED. SD.{RL, AQRL} is RV64-only.

The rl bit is mandatory because the two encodings that would be produced are not seen as useful at this time. The version with neither the aq nor the rl bit set would correspond to a store with no ordering annotations that was guaranteed to be performed atomically. This can be achieved with ordinary store instructions by suitably aligned pointers. The version with only the aq bit would correspond to store-acquire. Store-acquire has theoretical applications in seqlocks, but is not supported in language-level memory models and so is not included.

Chapter 18. RVWMO Memory Consistency Model, Version 2.0

This chapter defines the RISC-V memory consistency model. A memory consistency model is a set of rules specifying the values that can be returned by loads of memory. RISC-V uses a memory model called "RVWMO" (RISC-V Weak Memory Ordering) which is designed to provide flexibility for architects to build high-performance scalable designs while simultaneously supporting a tractable programming model.

Under RVWMO, code running on a single hart appears to execute in order from the perspective of other memory instructions in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts. The base RISC-V ISA provides a FENCE instruction for this purpose, described in Section 2.7, while the atomics extension "A" additionally defines load-reserved/store-conditional and atomic read-modify-write instructions.

The standard ISA extension for total store ordering "Ztso" (Chapter 19) augments RVWMO with additional rules specific to those extensions.

The appendices to this specification provide both axiomatic and operational formalizations of the memory consistency model as well as additional explanatory material.

This chapter defines the memory model for regular main memory operations. The interaction of the memory model with I/O memory, instruction fetches, FENCE.I, page-table walks, and SFENCE.VMA is not (yet) formalized. Some or all of the above may be formalized in a future revision of this specification. Future ISA extensions such as the V vector and J JIT extensions will need to be incorporated into a future revision as well.

Memory consistency models supporting overlapping memory accesses of different widths simultaneously remain an active area of academic research and are not yet fully understood. The specifics of how memory accesses of different sizes interact under RVWMO are specified to the best of our current abilities, but they are subject to revision should new issues be uncovered.

18.1. Definition of the RVWMO Memory Model

The RVWMO memory model is defined in terms of the *global memory order*, a total ordering of the memory operations produced by all harts. In general, a multithreaded program has many different possible executions, with each execution having its own corresponding global memory order.

The global memory order is defined over the primitive load and store operations generated by memory instructions. It is then subject to the constraints defined in the rest of this chapter. Any execution satisfying all of the memory model constraints is a legal execution (as far as the memory model is concerned).

18.1.1. Memory Model Primitives

The *program order* over memory operations reflects the order in which the instructions that generate each load and store are logically laid out in that hart's dynamic instruction stream; i.e., the order in which a simple in-order processor would execute the instructions of that hart.

Memory-accessing instructions give rise to *memory operations*. A memory operation can be either a *load operation*, a *store operation*, or both simultaneously. All memory operations are single-copy atomic: they can never be observed in a partially complete state.

Each aligned memory instruction that accesses XLEN or fewer bits gives rise to exactly one memory operation, unless specified otherwise. An aligned AMO gives rise to a single memory operation that is both a load operation and a store operation simultaneously.

Among instructions in RV32GC and RV64GC, the following are exceptions to the rule that an aligned memory instruction gives rise to exactly one memory operation:

- *An unsuccessful SC instruction does not give rise to any memory operations.*
- *Floating-point load and store instructions that access more than XLEN bits (e.g., FLD/FSD in RV32) may each give rise to multiple memory operations.*

ISA extensions such as **V** *(Vector) and the upcoming* **P** *(SIMD) may give rise to multiple memory operations. However, the memory model for these extensions has not yet been formalized.*

A misaligned load or store instruction may be decomposed into a set of component memory operations of any granularity. A floating-point load or store of more than XLEN bits may also be decomposed into a set of component memory operations of any granularity. The memory operations generated by such instructions are not ordered with respect to each other in program order, but they are ordered normally with respect to the memory operations generated by preceding and subsequent instructions in program order. The atomics extension "A" does not require execution environments to support misaligned atomic instructions at all. However, if misaligned atomics are supported via the misaligned atomicity granule PMA, then AMOs within an atomicity granule are not decomposed, nor are loads and stores defined in the base ISAs, nor are loads and stores of no more than XLEN bits defined in the F, D, and Q extensions.

The decomposition of misaligned memory operations down to byte granularity facilitates emulation on implementations that do not natively support misaligned accesses. Such implementations might, for example, simply iterate over the bytes of a misaligned access one by one.

An LR instruction and an SC instruction are said to be *paired* if the LR precedes the SC in program order and if there are no other LR or SC instructions in between; the corresponding memory operations are said to be paired as well (except in case of a failed SC, where no store operation is generated). The complete list of conditions determining whether an SC must succeed, may succeed, or must fail is defined in Section 13.2.

Load and store operations may also carry one or more ordering annotations from the following set: "acquire-RCpc", "acquire-RCsc", "release-RCpc", and "release-RCsc". An AMO or LR instruction with *aq* set has an "acquire-RCsc" annotation. An AMO or SC instruction with *rl* set has a "release-RCsc" annotation. An AMO, LR, or SC instruction with both *aq* and *rl* set has both "acquire-RCsc" and "release-RCsc" annotations.

For convenience, we use the term "acquire annotation" to refer to an acquire-RCpc annotation or an acquire-RCsc annotation. Likewise, a "release annotation" refers to a release-RCpc annotation or a release-RCsc annotation. An "RCpc annotation" refers to an acquire-RCpc annotation or a release-RCpc annotation. An *RCsc annotation* refers to an acquire-RCsc annotation or a release-RCsc annotation.

In the memory model literature, the term "RCpc" stands for release consistency with processor-consistent synchronization operations, and the term "RCsc" stands for release consistency with sequentially consistent synchronization operations.

While there are many different definitions for acquire and release annotations in the literature, in the context of RVWMO these terms are concisely and completely defined by Preserved Program Order rules 5-7.

"RCpc" annotations are currently only used when implicitly assigned to every memory access

> *per the standard extension "Ztso" (Chapter 19). Furthermore, although the ISA does not currently contain native load-acquire or store-release instructions, nor RCpc variants thereof, the RVWMO model itself is designed to be forwards-compatible with the potential addition of any or all of the above into the ISA in a future extension.*

18.1.2. Syntactic Dependencies

The definition of the RVWMO memory model depends in part on the notion of a syntactic dependency, defined as follows.

In the context of defining dependencies, a *register* refers either to an entire general-purpose register, some portion of a CSR, or an entire CSR. The granularity at which dependencies are tracked through CSRs is specific to each CSR and is defined in Section 18.2.

Syntactic dependencies are defined in terms of instructions' *source registers*, instructions' *destination registers*, and the way instructions *carry a dependency* from their source registers to their destination registers. This section provides a general definition of all of these terms; however, Section 18.3 provides a complete listing of the specifics for each instruction.

In general, a register *r* other than `x0` is a *source register* for an instruction *i* if any of the following hold:

- In the opcode of *i*, *rs1*, *rs2*, or *rs3* is set to *r*
- *i* is a CSR instruction, and in the opcode of *i*, *csr* is set to *r*, unless *i* is CSRRW or CSRRWI and *rd* is set to `x0`
- *r* is a CSR and an implicit source register for *i*, as defined in Section 18.3
- *r* is a CSR that aliases with another source register for *i*

Memory instructions also further specify which source registers are *address source registers* and which are *data source registers*.

In general, a register *r* other than `x0` is a *destination register* for an instruction *i* if any of the following hold:

- In the opcode of *i*, *rd* is set to *r*
- *i* is a CSR instruction, and in the opcode of *i*, *csr* is set to *r*, unless *i* is CSRRS or CSRRC and *rs1* is set to `x0` or *i* is CSRRSI or CSRRCI and uimm[4:0] is set to zero.
- *r* is a CSR and an implicit destination register for *i*, as defined in Section 18.3
- *r* is a CSR that aliases with another destination register for *i*

Most non-memory instructions *carry a dependency* from each of their source registers to each of their destination registers. However, there are exceptions to this rule; see Section 18.3.

Instruction *j* has a *syntactic dependency* on instruction *i* via destination register *s* of *i* and source register *r* of *j* if either of the following hold:

- *s* is the same as *r*, and no instruction program-ordered between *i* and *j* has *r* as a destination register
- There is an instruction *m* program-ordered between *i* and *j* such that all of the following hold:
  1. *j* has a syntactic dependency on *m* via destination register *q* and source register *r*
  2. *m* has a syntactic dependency on *i* via destination register *s* and source register *p*
  3. *m* carries a dependency from *p* to *q*

Finally, in the definitions that follow, let *a* and *b* be two memory operations, and let *i* and *j* be the instructions that generate *a* and *b*, respectively.

b has a *syntactic address dependency* on *a* if *r* is an address source register for *j* and *j* has a syntactic dependency on *i* via source register *r*

b has a *syntactic data dependency* on *a* if *b* is a store operation, *r* is a data source register for *j*, and *j* has a syntactic dependency on *i* via source register *r*

b has a *syntactic control dependency* on *a* if there is an instruction *m* program-ordered between *i* and *j* such that *m* is a branch or indirect jump and *m* has a syntactic dependency on *i*.

Generally speaking, non-AMO load instructions do not have data source registers, and unconditional non-AMO store instructions do not have destination registers. However, a successful SC instruction is considered to have the register specified in rd as a destination register, and hence it is possible for an instruction to have a syntactic dependency on a successful SC instruction that precedes it in program order.

18.1.3. Preserved Program Order

The global memory order for any given execution of a program respects some but not all of each hart's program order. The subset of program order that must be respected by the global memory order is known as *preserved program order*.

The complete definition of preserved program order is as follows (and note that AMOs are simultaneously both loads and stores): memory operation *a* precedes memory operation *b* in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program order, *a* and *b* both access regular main memory (rather than I/O regions), and any of the following hold:

- Overlapping-Address Orderings:
  1. *b* is a store, and *a* and *b* access overlapping memory addresses
  2. *a* and *b* are loads, *x* is a byte read by both *a* and *b*, there is no store to *x* between *a* and *b* in program order, and *a* and *b* return values for *x* written by different memory operations
  3. *a* is generated by an AMO or SC instruction, *b* is a load, and *b* returns a value written by *a*
- Explicit Synchronization
  4. There is a FENCE instruction that orders *a* before *b*
  5. *a* has an acquire annotation
  6. *b* has a release annotation
  7. *a* and *b* both have RCsc annotations
  8. *a* is paired with *b*
- Syntactic Dependencies
  9. *b* has a syntactic address dependency on *a*
  10. *b* has a syntactic data dependency on *a*
  11. *b* is a store, and *b* has a syntactic control dependency on *a*
- Pipeline Dependencies
  12. *b* is a load, and there exists some store *m* between *a* and *b* in program order such that *m* has an address or data dependency on *a*, and *b* returns a value written by *m*

13. *b* is a store, and there exists some instruction *m* between *a* and *b* in program order such that *m* has an address dependency on *a*

18.1.4. Memory Model Axioms

An execution of a RISC-V program obeys the RVWMO memory consistency model only if there exists a global memory order conforming to preserved program order and satisfying the *load value axiom*, the *atomicity axiom*, and the *progress axiom*.

18.1.4.1. Load Value Axiom

Each byte of each load *i* returns the value written to that byte by the store that is the latest in global memory order among the following stores:

1. Stores that write that byte and that precede *i* in the global memory order
2. Stores that write that byte and that precede *i* in program order

18.1.4.2. Atomicity Axiom

If *r* and *w* are paired load and store operations generated by aligned LR and SC instructions in a hart *h*, *s* is a store to byte *x*, and *r* returns a value written by *s*, then *s* must precede *w* in the global memory order, and there can be no store from a hart other than *h* to byte *x* following *s* and preceding *w* in the global memory order.

The Atomicity Axiom theoretically supports LR/SC pairs of different widths and to mismatched addresses, since implementations are permitted to allow SC operations to succeed in such cases. However, in practice, we expect such patterns to be rare, and their use is discouraged.

18.1.4.3. Progress Axiom

No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

18.2. CSR Dependency Tracking Granularity

Table 12. Granularities at which syntactic dependencies are tracked through CSRs

Name	Portions Tracked as Independent Units	Aliases
fflags	Bits 4, 3, 2, 1, 0	*fcsr*
frm	entire CSR	*fcsr*
fcsr	Bits 7-5, 4, 3, 2, 1, 0	*fflags*, *frm*

Note: read-only CSRs are not listed, as they do not participate in the definition of syntactic dependencies.

18.3. Source and Destination Register Listings

This section provides a concrete listing of the source and destination registers for each instruction. These listings are used in the definition of syntactic dependencies in Section 18.1.2.

The term "accumulating CSR" is used to describe a CSR that is both a source and a destination register, but

which carries a dependency only from itself to itself.

Instructions carry a dependency from each source register in the "Source Registers" column to each destination register in the "Destination Registers" column, from each source register in the "Source Registers" column to each CSR in the "Accumulating CSRs" column, and from each CSR in the "Accumulating CSRs" column to itself, except where annotated otherwise.

Key:

- [A]Address source register
- [D]Data source register
- † The instruction does not carry a dependency from any source register to any destination register
- ‡ The instruction carries dependencies from source register(s) to destination register(s) as specified

Table 13. RV32I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs	
LUI		*rd*		
AUIPC		*rd*		
JAL		*rd*		
JALR†	*rs1*	*rd*		
BEQ	*rs1*, *rs2*			
BNE	*rs1*, *rs2*			
BLT	*rs1*, *rs2*			
BGE	*rs1*, *rs2*			
BLTU	*rs1*, *rs2*			
BGEU	*rs1*, *rs2*			
LB †	*rs1* [A]	*rd*		
LH †	*rs1* [A]	*rd*		
LW †	*rs1* [A]	*rd*		
LBU †	*rs1* [A]	*rd*		
LHU †	*rs1* [A]	*rd*		
SB	*rs1* [A], *rs2* [D]			
SH	*rs1* [A], *rs2* [D]			
SW	*rs1* [A], *rs2* [D]			
ADDI	*rs1*	*rd*		
SLTI	*rs1*	*rd*		
SLTIU	*rs1*	*rd*		
XORI	*rs1*	*rd*		
ORI	*rs1*	*rd*		
ANDI	*rs1*	*rd*		
SLLI	*rs1*	*rd*		
SRLI	*rs1*	*rd*		
SRAI	*rs1*	*rd*		

	Source Registers	Destination Registers	Accumulating CSRs	
ADD	*rs1*, *rs2*	*rd*		
SUB	*rs1*, *rs2*	*rd*		
SLL	*rs1*, *rs2*	*rd*		
SLT	*rs1*, *rs2*	*rd*		
SLTU	*rs1*, *rs2*	*rd*		
XOR	*rs1*, *rs2*	*rd*		
SRL	*rs1*, *rs2*	*rd*		
SRA	*rs1*, *rs2*	*rd*		
OR	*rs1*, *rs2*	*rd*		
AND	*rs1*, *rs2*	*rd*		
FENCE				
FENCE.I				
ECALL				
EBREAK				
CSRRW‡	*rs1*, *csr**	*rd*, *csr*		*unless *rd*=`x0`
‡ carries a dependency from *rs1* to *csr* and from *csr* to *rd*				
CSRRS‡	*rs1*, *csr*	*rd*, *csr**		*unless *rs1*=`x0`
CSRRC‡	*rs1*, *csr*	*rd*, *csr**		*unless *rs1*=`x0`
‡ carries a dependency from *csr* and *rs1* to *csr* and from *csr* to *rd*				
CSRRWI ‡	*csr* *	*rd*, *csr*		*unless *rd=x0*
‡ carries a dependency from *csr* to *rd*				
CSRRSI ‡	*csr*	*rd*, *csr**		*unless uimm[4:0]=0
CSRRCI ‡	*csr*	*rd*, *csr**		*unless uimm[4:0]=0
‡ carries a dependency from *csr* to *rd* and *csr*				

Table 14. RV64I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs	
LWU †	*rs1* [A]	*rd*		
LD †	*rs1* [A]	*rd*		
SD	*rs1* [A], *rs2* [D]			
SLLI	*rs1*	*rd*		
SRLI	*rs1*	*rd*		
SRAI	*rs1*	*rd*		
ADDIW	*rs1*	*rd*		
SLLIW	*rs1*	*rd*		
SRLIW	*rs1*	*rd*		
SRAIW	*rs1*	*rd*		
ADDW	*rs1*, *rs2*	*rd*		
SUBW	*rs1*, *rs2*	*rd*		
SLLW	*rs1*, *rs2*	*rd*		

	Source Registers	Destination Registers	Accumulating CSRs	
SRLW	*rs1*, *rs2*	*rd*		
SRAW	*rs1*, *rs2*	*rd*		

Table 15. RV32M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
MUL	*rs1*, *rs2*	*rd*		
MULH	*rs1*, *rs2*	*rd*		
MULHSU	*rs1*, *rs2*	*rd*		
MULHU	*rs1*, *rs2*	*rd*		
DIV	*rs1*, *rs2*	*rd*		
DIVU	*rs1*, *rs2*	*rd*		
REM	*rs1*, *rs2*	*rd*		
REMU	*rs1*, *rs2*	*rd*		

Table 16. RV64M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
MULW	*rs1*, *rs2*	*rd*		
DIVW	*rs1*, *rs2*	*rd*		
DIVUW	*rs1*, *rs2*	*rd*		
REMW	*rs1*, *rs2*	*rd*		
REMUW	*rs1*, *rs2*	*rd*		

Table 17. RV32A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.W†	*rs1* ^{A}	*rd*		
SC.W†	*rs1* ^{A}, *rs2* ^{D}	*rd* *		* if successful
AMOSWAP.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOADD.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOXOR.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOAND.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOOR.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOMIN.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOMAX.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOMINU.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		
AMOMAXU.W†	*rs1* ^{A}, *rs2* ^{D}	*rd*		

Table 18. RV64A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.D†	*rs1* ^{A}	*rd*		
SC.D†	*rs1* ^{A}, *rs2* ^{D}	*rd* *		*if successful
AMOSWAP.D†	*rs1* ^{A}, *rs2* ^{D}	*rd*		

	Source Registers	Destination Registers	Accumulating CSRs	
AMOADD.D†	*rs1* [A], *rs2* [D]	*rd*		
AMOXOR.D†	*rs1* [A], *rs2* [D]	*rd*		
AMOAND.D†	*rs1* [A], *rs2*[D]	*rd*		
AMOOR.D†	*rs1* [A], *rs2*[D]	*rd*		
AMOMIN.D†	*rs1* [A], *rs2*[D]	*rd*		
AMOMAX.D†	*rs1* [A], *rs2*[D]	*rd*		
AMOMINU.D†	*rs1* [A], *rs2*[D]	*rd*		
AMOMAXU.D†	*rs1* [A], *rs2*[D]	*rd*		

Table 19. RV32F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FLW†	*rs1* [A]	*rd*		
FSW	*rs1* [A], *rs2*[D]			
FMADD.S	*rs1*, *rs2*, *rs3*, frm[*]	*rd*	NV, OF, UF, NX	[*]if rm=111
FMSUB.S	*rs1*, *rs2*, *rs3*, frm[*]	*rd*	NV, OF, UF, NX	[*]if rm=111
FNMSUB.S	*rs1*, *rs2*, *rs3*, frm[*]	*rd*	NV, OF, UF, NX	[*]if rm=111
FNMADD.S	*rs1*, *rs2*, *rs3*, frm[*]	*rd*	NV, OF, UF, NX	[*]if rm=111
FADD.S	*rs1*, *rs2*, frm[*]	*rd*	NV, OF, NX	[*]if rm=111
FSUB.S	*rs1*, *rs2*, frm[*]	*rd*	NV, OF, NX	[*]if rm=111
FMUL.S	*rs1*, *rs2*, frm[*]	*rd*	NV, OF, UF, NX	[*]if rm=111
FDIV.S	*rs1*, *rs2*, frm[*]	*rd*	NV, DZ, OF, UF, NX	[*]if rm=111
FSQRT.S	*rs1*, frm[*]	*rd*	NV, NX	[*]if rm=111
FSGNJ.S	*rs1*, *rs2*	*rd*		
FSGNJN.S	*rs1*, *rs2*	*rd*		
FSGNJX.S	*rs1*, *rs2*	*rd*		
FMIN.S	*rs1*, *rs2*	*rd*	NV	
FMAX.S	*rs1*, *rs2*	*rd*	NV	
FCVT.W.S	*rs1*, frm[*]	*rd*	NV, NX	[*]if rm=111
FCVT.WU.S	*rs1*, frm[*]	*rd*	NV, NX	[*]if rm=111
FMV.X.W	*rs1*	*rd*		
FEQ.S	*rs1*, *rs2*	*rd*	NV	
FLT.S	*rs1*, *rs2*	*rd*	NV	
FLE.S	*rs1*, *rs2*	*rd*	NV	
FCLASS.S	*rs1*	*rd*		
FCVT.S.W	*rs1*, frm[*]	*rd*	NX	[*]if rm=111
FCVT.S.WU	*rs1*, frm[*]	*rd*	NX	[*]if rm=111
FMV.W.X	*rs1*	*rd*		

Table 20. RV64F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.S	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.LU.S	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.S.L	*rs1*, frm*	*rd*	NX	*if rm=111
FCVT.S.LU	*rs1*, frm*	*rd*	NX	*if rm=111

Table 21. RV32D Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FLD†	*rs1*[A]	*rd*		
FSD	*rs1*[A], *rs2*[D]			
FMADD.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FMSUB.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMSUB.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FNMADD.D	*rs1*, *rs2*, *rs3*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FADD.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FSUB.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, NX	*if rm=111
FMUL.D	*rs1*, *rs2*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FDIV.D	*rs1*, *rs2*, frm*	*rd*	NV, DZ, OF, UF, NX	*if rm=111
FSQRT.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FSGNJ.D	*rs1*, *rs2*	*rd*		
FSGNJN.D	*rs1*, *rs2*	*rd*		
FSGNJX.D	*rs1*, *rs2*	*rd*		
FMIN.D	*rs1*, *rs2*	*rd*	NV	
FMAX.D	*rs1*, *rs2*	*rd*	NV	
FCVT.S.D	*rs1*, frm*	*rd*	NV, OF, UF, NX	*if rm=111
FCVT.D.S	*rs1*	*rd*	NV	
FEQ.D	*rs1*, *rs2*	*rd*	NV	
FLT.D	*rs1*, *rs2*	*rd*	NV	
FLE.D	*rs1*, *rs2*	*rd*	NV	
FCLASS.D	*rs1*	*rd*		
FCVT.W.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.WU.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.D.W	*rs1*	*rd*		
FCVT.D.WU	*rs1*	*rd*		

Table 22. RV64D Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FCVT.LU.D	*rs1*, frm*	*rd*	NV, NX	*if rm=111
FMV.X.D	*rs1*	*rd*		
FCVT.D.L	*rs1*, frm*	*rd*	NX	*if rm=111

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.D.LU	*rs1*, frm*	*rd*	NX	*if rm=111
FMV.D.X	*rs1*	*rd*		

Chapter 19. "Ztso" Extension for Total Store Ordering, Version 1.0

This chapter defines the "Ztso" extension for the RISC-V Total Store Ordering (RVTSO) memory consistency model. RVTSO is defined as a delta from RVWMO, which is defined in Section 18.1.

The Ztso extension is meant to facilitate the porting of code originally written for the x86 or SPARC architectures, both of which use TSO by default. It also supports implementations which inherently provide RVTSO behavior and want to expose that fact to software.

RVTSO makes the following adjustments to RVWMO:

- All load operations behave as if they have an acquire-RCpc annotation
- All store operations behave as if they have a release-RCpc annotation.
- All AMOs behave as if they have both acquire-RCsc and release-RCsc annotations.

These rules render all PPO rules except 4-7 redundant. They also make redundant any non-I/O fences that do not have both PW and SR set. Finally, they also imply that no memory operation will be reordered past an AMO in either direction.

In the context of RVTSO, as is the case for RVWMO, the storage ordering annotations are concisely and completely defined by PPO rules 5-7. In both of these memory models, it is the Section 18.1.4.1 that allows a hart to forward a value from its store buffer to a subsequent (in program order) load—that is to say that stores can be forwarded locally before they are visible to other harts.

Additionally, if the Ztso extension is implemented, then vector memory instructions in the V extension and Zve family of extensions follow RVTSO at the instruction level. The Ztso extension does not strengthen the ordering of intra-instruction element accesses.

In spite of the fact that Ztso adds no new instructions to the ISA, code written assuming RVTSO will not run correctly on implementations not supporting Ztso. Binaries compiled to run only under Ztso should indicate as such via a flag in the binary, so that platforms which do not implement Ztso can simply refuse to run them.

Chapter 20. "CMO" Extensions for Base Cache Management Operation ISA, Version 1.0.0

20.1. Pseudocode for instruction semantics

The semantics of each instruction in the Instructions chapter is expressed in a SAIL-like syntax.

20.2. Introduction

Cache-management operation (or *CMO*) instructions perform operations on copies of data in the memory hierarchy. In general, CMO instructions operate on cached copies of data, but in some cases, a CMO instruction may operate on memory locations directly. Furthermore, CMO instructions are grouped by operation into the following classes:

- A *management* instruction manipulates cached copies of data with respect to a set of agents that can access the data
- A *zero* instruction zeros out a range of memory locations, potentially allocating cached copies of data in one or more caches
- A *prefetch* instruction indicates to hardware that data at a given memory location may be accessed in the near future, potentially allocating cached copies of data in one or more caches

This document introduces a base set of CMO ISA extensions that operate specifically on cache blocks or the memory locations corresponding to a cache block; these are known as *cache-block operation* (or *CBO*) instructions. Each of the above classes of instructions represents an extension in this specification:

- The *Zicbom* extension defines a set of cache-block management instructions: `CBO.INVAL`, `CBO.CLEAN`, and `CBO.FLUSH`
- The *Zicboz* extension defines a cache-block zero instruction: `CBO.ZERO`
- The *Zicbop* extension defines a set of cache-block prefetch instructions: `PREFETCH.R`, `PREFETCH.W`, and `PREFETCH.I`

The execution behavior of the above instructions is also modified by CSR state added by this specification.

The remainder of this document provides general background information on CMO instructions and describes each of the above ISA extensions.

The term CMO encompasses all operations on caches or resources related to caches. The term CBO represents a subset of CMOs that operate only on cache blocks. The first CMO extensions only define CBOs.

20.3. Background

This chapter provides information common to all CMO extensions.

20.3.1. Memory and Caches

A *memory location* is a physical resource in a system uniquely identified by a *physical address*. An *agent* is a logic block, such as a RISC-V hart, accelerator, I/O device, etc., that can access a given memory location.

A given agent may not be able to access all memory locations in a system, and two different

agents may or may not be able to access the same set of memory locations.

A *load operation* (or *store operation*) is performed by an agent to consume (or modify) the data at a given memory location. Load and store operations are performed as a result of explicit memory accesses to that memory location. Additionally, a *read transfer* from memory fetches the data at the memory location, while a *write transfer* to memory updates the data at the memory location.

A *cache* is a structure that buffers copies of data to reduce average memory latency. Any number of caches may be interspersed between an agent and a memory location, and load and store operations from an agent may be satisfied by a cache instead of the memory location.

Load and store operations are decoupled from read and write transfers by caches. For example, a load operation may be satisfied by a cache without performing a read transfer from memory, or a store operation may be satisfied by a cache that first performs a read transfer from memory.

Caches organize copies of data into *cache blocks*, each of which represents a contiguous, naturally aligned power-of-two (or *NAPOT*) range of memory locations. A cache block is identified by any of the physical addresses corresponding to the underlying memory locations. The capacity and organization of a cache and the size of a cache block are both *implementation-specific*, and the execution environment provides software a means to discover information about the caches and cache blocks in a system. In the initial set of CMO extensions, the size of a cache block shall be uniform throughout the system.

In future CMO extensions, the requirement for a uniform cache block size may be relaxed.

Implementation techniques such as speculative execution or hardware prefetching may cause a given cache to allocate or deallocate a copy of a cache block at any time, provided the corresponding physical addresses are accessible according to the supported access type PMA and are cacheable according to the cacheability PMA. Allocating a copy of a cache block results in a read transfer from another cache or from memory, while deallocating a copy of a cache block may result in a write transfer to another cache or to memory depending on whether the data in the copy were modified by a store operation. Additional details are discussed in Coherent Agents and Caches.

20.3.2. Cache-Block Operations

A CBO instruction causes one or more operations to be performed on the cache blocks identified by the instruction. In general, a CBO instruction may identify one or more cache blocks; however, in the initial set of CMO extensions, CBO instructions identify a single cache block only.

A cache-block management instruction performs one of the following operations, relative to the copy of a given cache block allocated in a given cache:

- An *invalidate operation* deallocates the copy of the cache block
- A *clean operation* performs a write transfer to another cache or to memory if the data in the copy of the cache block have been modified by a store operation
- A *flush operation* atomically performs a clean operation followed by an invalidate operation

Additional details, including the actual operation performed by a given cache-block management instruction, are described in Cache-Block Management Instructions.

A cache-block zero instruction performs a set of store operations that write zeros to the set of bytes corresponding to a cache block. Unless specified otherwise, the store operations generated by a cache-block zero instruction have the same general properties and behaviors that other store instructions in the architecture have. An implementation may or may not update the entire set of bytes atomically with a

single store operation. Additional details are described in Cache-Block Zero Instructions.

A cache-block prefetch instruction is a HINT to the hardware that software expects to perform a particular type of memory access in the near future. Additional details are described in Cache-Block Prefetch Instructions.

20.4. Coherent Agents and Caches

For a given memory location, a *set of coherent agents* consists of the agents for which all of the following hold:

- Store operations from all agents in the set appear to be serialized with respect to each other
- Store operations from all agents in the set eventually appear to all other agents in the set
- A load operation from an agent in the set returns data from a store operation from an agent in the set (or from the initial data in memory)

The coherent agents within such a set shall access a given memory location with the same physical address and the same physical memory attributes; however, if the coherence PMA for a given agent indicates a given memory location is not coherent, that agent shall not be a member of a set of coherent agents with any other agent for that memory location and shall be the sole member of a set of coherent agents consisting of itself.

An agent who is a member of a set of coherent agents is said to be *coherent* with respect to the other agents in the set. On the other hand, an agent who is *not* a member is said to be *non-coherent* with respect to the agents in the set.

Caches introduce the possibility that multiple copies of a given cache block may be present in a system at the same time. An *implementation-specific* mechanism keeps these copies coherent with respect to the load and store operations from the agents in the set of coherent agents. Additionally, if a coherent agent in the set executes a CBO instruction that specifies the cache block, the resulting operation shall apply to any and all of the copies in the caches that can be accessed by the load and store operations from the coherent agents.

An operation from a CBO instruction is defined to operate only on the copies of a cache block that are cached in the caches accessible by the explicit memory accesses performed by the set of coherent agents. This includes copies of a cache block in caches that are accessed only indirectly by load and store operations, e.g. coherent instruction caches.

The set of caches subject to the above mechanism form a *set of coherent caches*, and each coherent cache has the following behaviors, assuming all operations are performed by the agents in a set of coherent agents:

- A coherent cache is permitted to allocate and deallocate copies of a cache block and perform read and write transfers as described in Memory and Caches
- A coherent cache is permitted to perform a write transfer to memory provided that a store operation has modified the data in the cache block since the most recent invalidate, clean, or flush operation on the cache block
- At least one coherent cache is responsible for performing a write transfer to memory once a store operation has modified the data in the cache block until the next invalidate, clean, or flush operation on the cache block, after which no coherent cache is responsible (or permitted) to perform a write transfer to memory until the next store operation has modified the data in the cache block
- A coherent cache is required to perform a write transfer to memory if a store operation has modified

the data in the cache block since the most recent invalidate, clean, or flush operation on the cache block and if the next clean or flush operation requires a write transfer to memory

The above restrictions ensure that a "clean" copy of a cache block, fetched by a read transfer from memory and unmodified by a store operation, cannot later overwrite the copy of the cache block in memory updated by a write transfer to memory from a non-coherent agent.

A non-coherent agent may initiate a cache-block operation that operates on the set of coherent caches accessed by a set of coherent agents. The mechanism to perform such an operation is *implementation-specific*.

20.4.1. Memory Ordering

20.4.1.1. Preserved Program Order

The preserved program order (abbreviated *PPO*) rules are defined by the RVWMO memory ordering model. How the operations resulting from CMO instructions fit into these rules is described below.

For cache-block management instructions, the resulting invalidate, clean, and flush operations behave as stores in the PPO rules subject to one additional overlapping address rule. Specifically, if *a* precedes *b* in program order, then *a* will precede *b* in the global memory order if:

- *a* is an invalidate, clean, or flush, *b* is a load, and *a* and *b* access overlapping memory addresses

The above rule ensures that a subsequent load in program order never appears in the global memory order before a preceding invalidate, clean, or flush operation to an overlapping address.

Additionally, invalidate, clean, and flush operations are classified as W or O (depending on the physical memory attributes for the corresponding physical addresses) for the purposes of predecessor and successor sets in `FENCE` instructions. These operations are *not* ordered by other instructions that order stores, e.g. `FENCE.I` and `SFENCE.VMA`.

For cache-block zero instructions, the resulting store operations behave as stores in the PPO rules and are ordered by other instructions that order stores.

Finally, for cache-block prefetch instructions, the resulting operations are *not* ordered by the PPO rules nor are they ordered by any other ordering instructions.

20.4.1.2. Load Values

An invalidate operation may change the set of values that can be returned by a load. In particular, an additional condition is added to the Load Value Axiom:

- If an invalidate operation *i* precedes a load *r* and operates on a byte *x* returned by *r*, and no store to *x* appears between *i* and *r* in program order or in the global memory order, then *r* returns any of the following values for *x*:
  1. If no clean or flush operations on *x* precede *i* in the global memory order, either the initial value of *x* or the value of any store to *x* that precedes *i*
  2. If no store to *x* precedes a clean or flush operation on *x* in the global memory order and if the clean or flush operation on *x* precedes *i* in the global memory order, either the initial value of *x* or the value of any store to *x* that precedes *i*

3. If a store to *x* precedes a clean or flush operation on *x* in the global memory order and if the clean or flush operation on *x* precedes *i* in the global memory order, either the value of the latest store to *x* that precedes the latest clean or flush operation on *x* or the value of any store to *x* that both precedes *i* and succeeds the latest clean or flush operation on *x* that precedes *i*
4. The value of any store to *x* by a non-coherent agent regardless of the above conditions

The first three bullets describe the possible load values at different points in the global memory order relative to clean or flush operations. The final bullet implies that the load value may be produced by a non-coherent agent at any time.

20.4.2. Traps

Execution of certain CMO instructions may result in traps due to CSR state, described in the Control and Status Register State section, or due to the address translation and protection mechanisms. The trapping behavior of CMO instructions is described in the following sections.

20.4.2.1. Illegal-Instruction and Virtual-Instruction Exceptions

Cache-block management instructions and cache-block zero instructions may raise illegal-instruction exceptions or virtual-instruction exceptions depending on the current privilege mode and the state of the CMO control registers described in the Control and Status Register State section.

Cache-block prefetch instructions raise neither illegal-instruction exceptions nor virtual-instruction exceptions.

20.4.2.2. Page-Fault, Guest-Page-Fault, and Access-Fault Exceptions

Similar to load and store instructions, CMO instructions are explicit memory access instructions that compute an effective address. The effective address is ultimately translated into a physical address based on the privilege mode and the enabled translation mechanisms, and the CMO extensions impose the following constraints on the physical addresses in a given cache block:

- The PMP access control bits shall be the same for *all* physical addresses in the cache block, and if write permission is granted by the PMP access control bits, read permission shall also be granted
- The PMAs shall be the same for *all* physical addresses in the cache block, and if write permission is granted by the supported access type PMAs, read permission shall also be granted

If the above constraints are not met, the behavior of a CBO instruction is UNSPECIFIED.

This specification assumes that the above constraints will typically be met for main memory regions and may be met for certain I/O regions.

The access size for CMO instructions is equal to the size of the cache block, however in some cases that access can be decomposed into multiple memory operations. PMP checks are applied to each memory operation independently. For example a 64-byte **cbo.zero** *that spans two 32-byte PMP regions would succeed if it was decomposed into two 32-byte memory operations (and the PMP access control bits are the same in both regions), but if performed as a single 64-byte memory operation it would cause an access fault.*

The Zicboz extension introduces an additional supported access type PMA for cache-block zero instructions. Main memory regions are required to support accesses by cache-block zero instructions; however, I/O regions may specify whether accesses by cache-block zero instructions are supported.

A cache-block management instruction is permitted to access the specified cache block whenever a load instruction or store instruction is permitted to access the corresponding physical addresses. If neither a load instruction nor store instruction is permitted to access the physical addresses, but an instruction fetch is permitted to access the physical addresses, whether a cache-block management instruction is permitted to access the cache block is UNSPECIFIED. If access to the cache block is not permitted, a cache-block management instruction raises a store page-fault or store guest-page-fault exception if address translation does not permit any access or raises a store access-fault exception otherwise. During address translation, the instruction also checks the accessed bit and may either raise an exception or set the bit as required.

The interaction between cache-block management instructions and instruction fetches will be specified in a future extension.

As implied by omission, a cache-block management instruction does not check the dirty bit and neither raises an exception nor sets the bit.

A cache-block zero instruction is permitted to access the specified cache block whenever a store instruction is permitted to access the corresponding physical addresses and when the PMAs indicate that cache-block zero instructions are a supported access type. If access to the cache block is not permitted, a cache-block zero instruction raises a store page-fault or store guest-page-fault exception if address translation does not permit write access or raises a store access-fault exception otherwise. During address translation, the instruction also checks the accessed and dirty bits and may either raise an exception or set the bits as required.

A cache-block prefetch instruction is permitted to access the specified cache block whenever a load instruction, store instruction, or instruction fetch is permitted to access the corresponding physical addresses. If access to the cache block is not permitted, a cache-block prefetch instruction does not raise any exceptions and shall not access any caches or memory. During address translation, the instruction does *not* check the accessed and dirty bits and neither raises an exception nor sets the bits.

When a page-fault, guest-page-fault, or access-fault exception is taken, the relevant *tval CSR is written with the faulting effective address (i.e. the value of *rs1*).

Like a load or store instruction, a CMO instruction may or may not be permitted to access a cache block based on the states of the `MPRV`, `MPV`, and `MPP` bits in `mstatus` and the `SUM` and `MXR` bits in `mstatus`, `sstatus`, and `vsstatus`.

This specification expects that implementations will process cache-block management instructions like store/AMO instructions, so store/AMO exceptions are appropriate for these instructions, regardless of the permissions required.

20.4.2.3. Address-Misaligned Exceptions

CMO instructions do *not* generate address-misaligned exceptions.

20.4.2.4. Breakpoint Exceptions and Debug Mode Entry

Unless otherwise defined by the debug architecture specification, the behavior of trigger modules with respect to CMO instructions is UNSPECIFIED.

For the Zicbom, Zicboz, and Zicbop extensions, this specification recommends the following common trigger module behaviors:

- *Type 6 address match triggers, i.e. `tdata1.type=6` and `mcontrol6.select=0`, should be supported*

- *Type 2 address/data match triggers, i.e.* `tdata1.type=2`*, should be unsupported*
- *The size of a memory access equals the size of the cache block accessed, and the compare values follow from the addresses of the NAPOT memory region corresponding to the cache block containing the effective address*
- *Unless an encoding for a cache block is added to the* `mcontrol6.size` *field, an address trigger should only match a memory access from a CBO instruction if* `mcontrol6.size=0`

If the Zicbom extension is implemented, this specification recommends the following additional trigger module behaviors:

- *Implementing address match triggers should be optional*
- *Type 6 data match triggers, i.e.* `tdata1.type=6` *and* `mcontrol6.select=1`*, should be unsupported*
- *Memory accesses are considered to be stores, i.e. an address trigger matches only if* `mcontrol6.store=1`

If the Zicboz extension is implemented, this specification recommends the following additional trigger module behaviors:

- *Implementing address match triggers should be mandatory*
- *Type 6 data match triggers, i.e.* `tdata1.type=6` *and* `mcontrol6.select=1`*, should be supported, and implementing these triggers should be optional*
- *Memory accesses are considered to be stores, i.e. an address trigger matches only if* `mcontrol6.store=1`

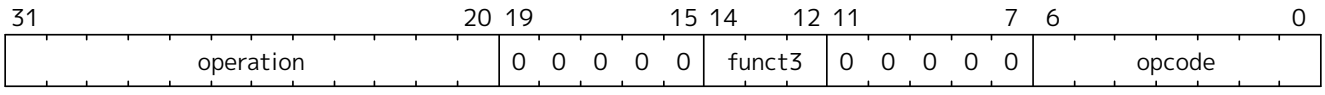
If the Zicbop extension is implemented, this specification recommends the following additional trigger module behaviors:

- *Implementing address match triggers should be optional*
- *Type 6 data match triggers, i.e.* `tdata1.type=6` *and* `mcontrol6.select=1`*, should be unsupported*
- *Memory accesses may be considered to be loads or stores depending on the implementation, i.e. whether an address trigger matches on these instructions when* `mcontrol6.load=1` *or* `mcontrol6.store=1` *is implementation-specific*

This specification also recommends that the behavior of trigger modules with respect to the Zicboz extension should be defined in version 1.0 of the debug architecture specification. The behavior of trigger modules with respect to the Zicbom and Zicbop extensions is expected to be defined in future extensions.

20.4.2.5. Hypervisor Extension

For the purposes of writing the `mtinst` or `htinst` register on a trap, the following standard transformation is defined for cache-block management instructions and cache-block zero instructions:



The `operation` field corresponds to the 12 most significant bits of the trapping instruction.

As described in the hypervisor extension, a zero may be written into `mtinst` *or* `htinst` *instead*

> *of the standard transformation defined above.*

20.4.3. Effects on Constrained LR/SC Loops

The following event is added to the list of events that satisfy the eventuality guarantee provided by constrained LR/SC loops, as defined in the A extension:

- Some other hart executes a cache-block management instruction or a cache-block zero instruction to the reservation set of the LR instruction in *H*'s constrained LR/SC loop.

> *The above event has been added to accommodate cache coherence protocols that cannot distinguish between invalidations for stores and invalidations for cache-block management operations.*
>
> *Aside from the above event, CMO instructions neither change the properties of constrained LR/SC loops nor modify the eventuality guarantee provided by them. For example, executing a CMO instruction may cause a constrained LR/SC loop on any hart to fail periodically or may cause a unconstrained LR/SC sequence on the same hart to fail always. Additionally, executing a cache-block prefetch instruction does not impact the eventuality guarantee provided by constrained LR/SC loops executed on any hart.*

20.4.4. Software Discovery

The initial set of CMO extensions requires the following information to be discovered by software:

- The size of the cache block for management and prefetch instructions
- The size of the cache block for zero instructions
- CBIE support at each privilege level

Other general cache characteristics may also be specified in the discovery mechanism.

20.5. CSR controls for CMO instructions

The xenvcfg registers control CBO instruction execution based on the current privilege mode and the state of the appropriate CSRs, as detailed below.

A `CBO.INVAL` instruction executes or raises either an illegal-instruction exception or a virtual-instruction exception based on the state of the `xenvcfg.CBIE` fields:

```
// illegal-instruction exceptions
if (((priv_mode != M) && (menvcfg.CBIE == 00)) ||
    ((priv_mode == U) && (senvcfg.CBIE == 00)))
{
  <raise illegal-instruction exception>
}
// virtual-instruction exceptions
else if (((priv_mode == VS) && (henvcfg.CBIE == 00)) ||
         ((priv_mode == VU) && ((henvcfg.CBIE == 00) || (senvcfg.CBIE == 00))))
{
  <raise virtual-instruction exception>
```

```
}
// execute instruction
else
{
  if (((priv_mode != M) && (menvcfg.CBIE == 01)) ||
      ((priv_mode == U) && (senvcfg.CBIE == 01)) ||
      ((priv_mode == VS) && (henvcfg.CBIE == 01)) ||
      ((priv_mode == VU) && ((henvcfg.CBIE == 01) || (senvcfg.CBIE == 01))))
  {
    <execute CBO.INVAL and perform flush operation>
  }
  else
  {
    <execute CBO.INVAL and perform invalidate operation>
  }
}
```

Until a modified cache block has updated memory, a `CBO.INVAL` instruction may expose stale data values in memory if the CSRs are programmed to perform an invalidate operation. This behavior may result in a security hole if lower privileged level software performs an invalidate operation and accesses sensitive information in memory.

To avoid such holes, higher privileged level software must perform either a clean or flush operation on the cache block before permitting lower privileged level software to perform an invalidate operation on the block. Alternatively, higher privileged level software may program the CSRs so that `CBO.INVAL` either traps or performs a flush operation in a lower privileged level.

A `CBO.CLEAN` or `CBO.FLUSH` instruction executes or raises an illegal-instruction or virtual-instruction exception based on the state of the `xenvcfg.CBCFE` bits:

```
// illegal-instruction exceptions
if (((priv_mode != M) && !menvcfg.CBCFE) ||
    ((priv_mode == U) && !senvcfg.CBCFE))
{
  <raise illegal-instruction exception>
}
// virtual-instruction exceptions
else if (((priv_mode == VS) && !henvcfg.CBCFE) ||
         ((priv_mode == VU) && !(henvcfg.CBCFE && senvcfg.CBCFE)))
{
  <raise virtual-instruction exception>
}
// execute instruction
else
{
  <execute CBO.CLEAN or CBO.FLUSH>
}
```

Finally, a `CBO.ZERO` instruction executes or raises an illegal-instruction or virtual-instruction exception based on the state of the `xenvcfg.CBZE` bits:

```
// illegal-instruction exceptions
if (((priv_mode != M) && !menvcfg.CBZE) ||
    ((priv_mode == U) && !senvcfg.CBZE))
{
  <raise illegal-instruction exception>
}
// virtual-instruction exceptions
else if (((priv_mode == VS) && !henvcfg.CBZE) ||
         ((priv_mode == VU) && !(henvcfg.CBZE && senvcfg.CBZE)))
{
  <raise virtual-instruction exception>
}
// execute instruction
else
{
  <execute CBO.ZERO>
}
```

The CBIE/CBCFE/CBZE fields in each `xenvcfg` register do not affect the read and write behavior of the same fields in the other `xenvcfg` registers.

Each `xenvcfg` register is WARL; however, software should determine the legal values from the execution environment discovery mechanism.

20.6. Extensions

CMO instructions are defined in the following extensions:

- Cache-Block Management Instructions
- Cache-Block Zero Instructions
- Cache-Block Prefetch Instructions

20.6.1. Cache-Block Management Instructions

Cache-block management instructions enable software running on a set of coherent agents to communicate with a set of non-coherent agents by performing one of the following operations:

- An invalidate operation makes data from store operations performed by a set of non-coherent agents visible to the set of coherent agents at a point common to both sets by deallocating all copies of a cache block from the set of coherent caches up to that point
- A clean operation makes data from store operations performed by the set of coherent agents visible to a set of non-coherent agents at a point common to both sets by performing a write transfer of a copy of a cache block to that point provided a coherent agent performed a store operation that modified the data in the cache block since the previous invalidate, clean, or flush operation on the cache block
- A flush operation atomically performs a clean operation followed by an invalidate operation

In the Zicbom extension, the instructions operate to a point common to *all* agents in the system. In other words, an invalidate operation ensures that store operations from all non-coherent agents visible to agents in the set of coherent agents, and a clean operation ensures that store operations from coherent agents visible to all non-coherent agents.

The Zicbom extension does not prohibit agents that fall outside of the above architectural definition; however, software cannot rely on the defined cache operations to have the desired effects with respect to those agents.

Future extensions may define different sets of agents for the purposes of performance optimization.

These instructions operate on the cache block whose effective address is specified in *rs1*. The effective address is translated into a corresponding physical address by the appropriate translation mechanisms.

The following instructions comprise the Zicbom extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	cbo.clean *base*	Cache Block Clean
✓	✓	cbo.flush *base*	Cache Block Flush
✓	✓	cbo.inval *base*	Cache Block Invalidate

Cache-block management instructions ignore cacheability attributes and operate on the cache block irrespective of the PMA cacheable attribute and any Page-Based Memory Type (PBMT) downgrade from cacheable to non-cacheable.

20.6.2. Cache-Block Zero Instructions

Cache-block zero instructions store zeros to the set of bytes corresponding to a cache block. An implementation may update the bytes in any order and with any granularity and atomicity, including individual bytes.

Cache-block zero instructions store zeros independently of whether data from the underlying memory locations are cacheable. In addition, this specification does not constrain how the bytes are written.

These instructions operate on the cache block, or the memory locations corresponding to the cache block, whose effective address is specified in *rs1*. The effective address is translated into a corresponding physical address by the appropriate translation mechanisms.

The following instructions comprise the Zicboz extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	cbo.zero *base*	Cache Block Zero

20.6.3. Cache-Block Prefetch Instructions

Cache-block prefetch instructions are HINTs to the hardware to indicate that software intends to perform a particular type of memory access in the near future. The types of memory accesses are instruction fetch, data read (i.e. load), and data write (i.e. store).

These instructions operate on the cache block whose effective address is the sum of the base address

specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* shall equal `0b00000`. The effective address is translated into a corresponding physical address by the appropriate translation mechanisms.

Cache-block prefetch instructions are encoded as ORI instructions with rd equal to `0b00000`; however, for the purposes of effective address calculation, this field is also interpreted as imm[4:0] like a store instruction.

The following instructions comprise the Zicbop extension:

RV32	RV64	Mnemonic	Instruction
✓	✓	prefetch.i *offset*(*base*)	Cache Block Prefetch for Instruction Fetch
✓	✓	prefetch.r *offset*(*base*)	Cache Block Prefetch for Data Read
✓	✓	prefetch.w *offset*(*base*)	Cache Block Prefetch for Data Write

20.7. Instructions

20.7.1. cbo.clean

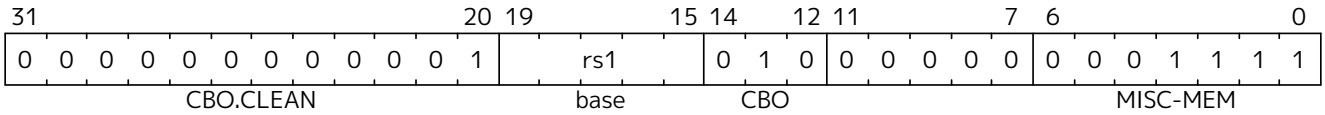
Synopsis

Perform a clean operation on a cache block

Mnemonic

cbo.clean *offset*(*base*)

Encoding



Description

A **cbo.clean** instruction performs a clean operation on the cache block whose effective address is the base address specified in *rs1*. The offset operand may be omitted; otherwise, any expression that computes the offset shall evaluate to zero. The instruction operates on the set of coherent caches accessed by the agent executing the instruction.

*When executing a **cbo.clean** instruction, an implementation may instead perform a flush operation, since the result of that operation is indistinguishable from the sequence of performing a clean operation just before deallocating all cached copies in the set of coherent caches.*

20.7.2. cbo.flush

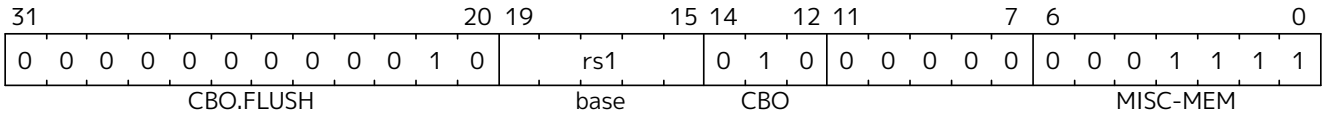
Synopsis

Perform a flush operation on a cache block

Mnemonic

cbo.flush *offset*(*base*)

Encoding



Description

A **cbo.flush** instruction performs a flush operation on the cache block whose that contains the address specified in *rs1*. It is not required that *rs1* is aligned to the size of a cache block. On faults, the faulting virtual address is considered to be the value in rs1, rather than the base address of the cache block. The instruction operates on the set of coherent caches accessed by the agent executing the instruction.

The assembly *offset* operand may be omitted. If it isn't then any expression that computes the offset shall evaluate to zero.

20.7.3. cbo.inval

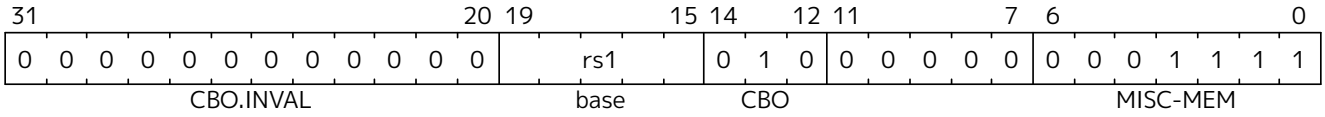
Synopsis

Perform an invalidate operation on a cache block

Mnemonic

cbo.inval *offset*(*base*)

Encoding



Description

A **cbo.inval** instruction performs an invalidate operation on the cache block that contains the address specified in *rs1*. It is not required that *rs1* is aligned to the size of a cache block. On faults, the faulting virtual address is considered to be the value in rs1, rather than the base address of the cache block. The instruction operates on the set of coherent caches accessed by the agent executing the instruction.

Depending on CSR programming, the instruction may perform a flush operation instead of an invalidate operation.

The assembly *offset* operand may be omitted. If it isn't then any expression that computes the offset shall evaluate to zero.

*When executing a **cbo.inval** instruction, an implementation may instead perform a flush operation, since the result of that operation is indistinguishable from the sequence of performing a write transfer to memory just before performing an invalidate operation.*

20.7.4. cbo.zero

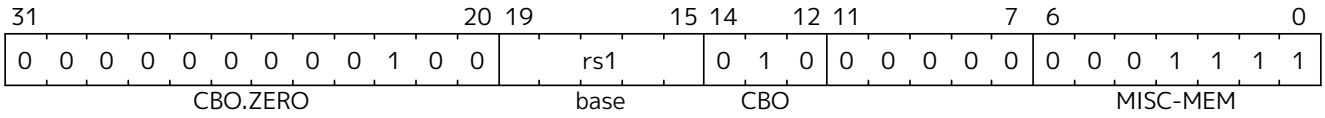
Synopsis

Store zeros to the full set of bytes corresponding to a cache block

Mnemonic

cbo.zero *offset*(*base*)

Encoding



Description

A **cbo.zero** instruction performs stores of zeros to the full set of bytes corresponding to the cache block that contains the address specified in *rs1*. It is not required that *rs1* is aligned to the size of a cache block. On faults, the faulting virtual address is considered to be the value in rs1, rather than the base address of the cache block. An implementation may or may not update the entire set of bytes atomically.

The assembly *offset* operand may be omitted. If it isn't then any expression that computes the offset shall evaluate to zero.

20.7.5. prefetch.i

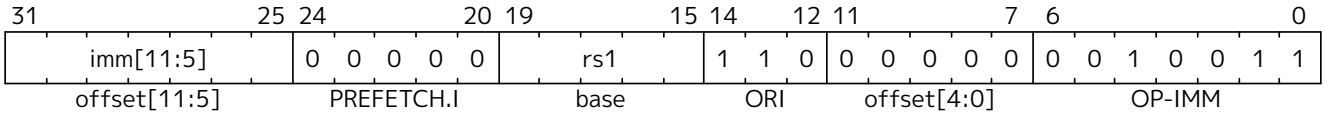
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by an instruction fetch in the near future

Mnemonic

prefetch.i *offset*(*base*)

Encoding



Description

A **prefetch.i** instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals `0b00000`, is likely to be accessed by an instruction fetch in the near future.

An implementation may opt to cache a copy of the cache block in a cache accessed by an instruction fetch in order to improve memory access latency, but this behavior is not required.

20.7.6. prefetch.r

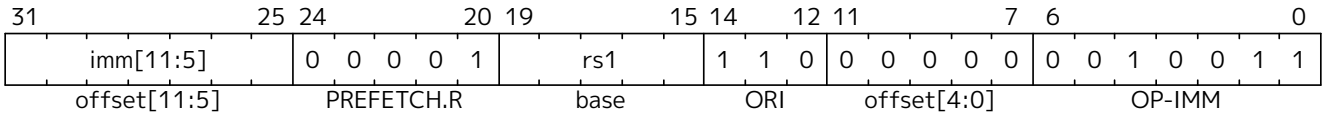
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data read in the near future

Mnemonic

prefetch.r *offset*(*base*)

Encoding



Description

A **prefetch.r** instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals `0b00000`, is likely to be accessed by a data read (i.e. load) in the near future.

An implementation may opt to cache a copy of the cache block in a cache accessed by a data read in order to improve memory access latency, but this behavior is not required.

20.7.7. prefetch.w

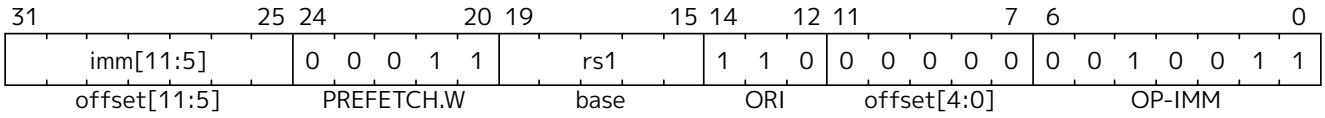
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data write in the near future

Mnemonic

prefetch.w *offset*(*base*)

Encoding



Description

A **prefetch.w** instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals `0b00000`, is likely to be accessed by a data write (i.e. store) in the near future.

An implementation may opt to cache a copy of the cache block in a cache accessed by a data write in order to improve memory access latency, but this behavior is not required.

Chapter 21. "F" Extension for Single-Precision Floating-Point, Version 2.2

This chapter describes the standard instruction-set extension for single-precision floating-point, which is named "F" and adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard [.citation](*ANSI/IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008*). The F extension depends on the "Zicsr" extension for control and status register access.

21.1. F Register State

The F extension adds 32 floating-point registers, `f0-f31`, each 32 bits wide, and a floating-point control and status register `fcsr`, which contains the operating mode and exception status of the floating-point unit. This additional state is shown in Table 23. We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, and FLEN=32 for the F single-precision floating-point extension. Most floating-point instructions operate on values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.

We considered a unified register file for both integer and floating-point values as this simplifies software register allocation and calling conventions, and reduces total user state. However, a split organization increases the total number of registers accessible with a given instruction width, simplifies provision of enough register file ports for wide superscalar issue, supports decoupled floating-point-unit architectures, and simplifies use of internal floating-point encoding techniques. Compiler support and calling conventions for split register file architectures are well understood, and using dirty bits on floating-point register file state can reduce context-switch overhead.

Table 23. RISC-V standard F extension single-precision floating-point state

FLEN-1	0
f0	
f1	
f2	
f3	
f4	
f5	
f6	
f7	
f8	
f9	
f10	
f11	
f12	
f13	
f14	
f15	
f16	
f17	
f18	
f19	

FLEN-1	0
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	
FLEN	
31	0
fcsr	
32	

21.2. Floating-Point Control and Status Register

The floating-point control and status register, `fcsr`, is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in Floating-Point Control and Status Register.

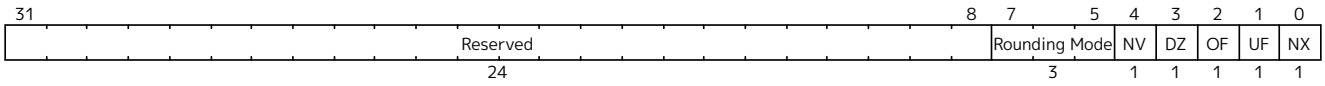


Figure 2. Floating-point control and status register

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads `fcsr` by copying it into integer register *rd*. FSCSR swaps the value in `fcsr` by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into `fcsr`.

The fields within the `fcsr` can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field `frm` (`fcsr` bits 7—5) and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in `frm` by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into `frm`. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field `fflags` (`fcsr` bits 4—0).

Bits 31—8 of the `fcsr` are reserved for other standard extensions. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in Table 24. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111).

Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

Table 24. Rounding mode encoding.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards −∞)
011	RUP	Round Up (towards +∞)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Reserved for future use.*
110		*Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.

The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

The ratified version of the F spec mandated that an illegal-instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal-instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 25. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 25. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

21.3. NaN Generation and Propagation

Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN.

The canonical NaN has a positive sign and all significand bits clear except the MSB, a.k.a. the quiet bit. For single-precision floating-point, this corresponds to the pattern `0x7fc00000`.

We considered propagating NaN payloads, as is recommended by the standard, but this decision would have increased hardware cost. Moreover, since this feature is optional in the standard, it cannot be used in portable code.

Implementers are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.

We require implementations to return the standard-mandated default values in the case of exceptional conditions, without any further intervention on the part of user-level software (unlike the Alpha ISA floating-point trap barriers). We believe full hardware handling of exceptional cases will become more common, and so wish to avoid complicating the user-level ISA to optimize other approaches. Implementations can always trap to machine-mode software handlers to provide exceptional default values.

21.4. Subnormal Arithmetic

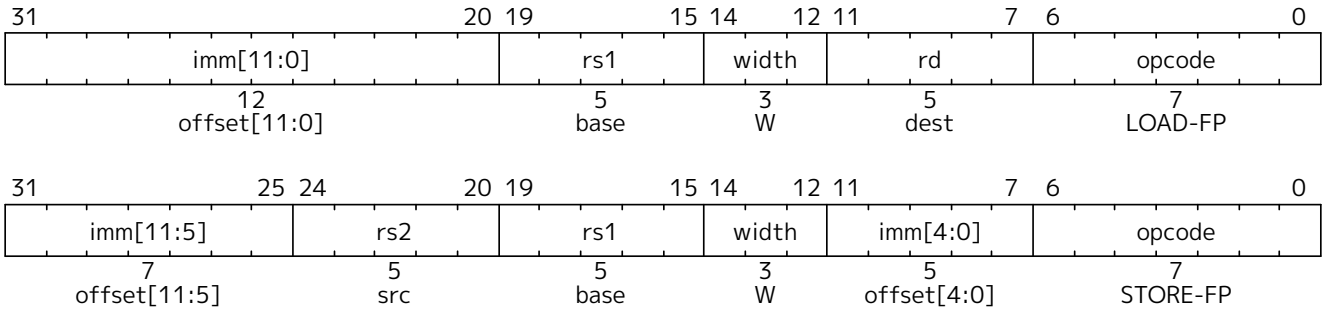
Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

In the parlance of the IEEE standard, tininess is detected after rounding.

Detecting tininess after rounding results in fewer spurious underflow signals.

21.5. Single-Precision Load and Store Instructions

Floating-point loads and stores use the same base+offset addressing mode as the integer base ISAs, with a base address in register *rs1* and a 12-bit signed byte offset. The FLW instruction loads a single-precision floating-point value from memory into floating-point register *rd*. FSW stores a single-precision value from floating-point register *rs2* to memory.



FLW and FSW are only guaranteed to execute atomically if the effective address is naturally aligned.

FLW and FSW do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

As described in Section 2.6, the execution environment defines whether misaligned floating-point loads and stores are handled invisibly or raise a contained or fatal trap.

21.6. Single-Precision Floating-Point Computational Instructions

Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode. FADD.S and FMUL.S perform single-precision floating-point addition and multiplication respectively, between *rs1* and *rs2*. FSUB.S performs the single-precision floating-point subtraction of *rs2* from *rs1*. FDIV.S performs the single-precision floating-point division of *rs1* by *rs2*. FSQRT.S computes the square root of *rs1*. In each case, the result is written to *rd*.

The 2-bit floating-point format field *fmt* is encoded as shown in Table 26. It is set to *S* (00) for all instructions in the F extension.

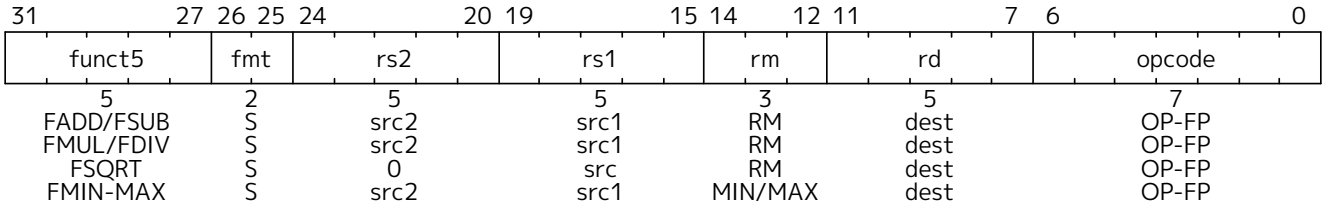
Table 26. Format field encoding

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in Table 24.

Floating-point minimum-number and maximum-number instructions FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. For the purposes of these instructions only, the value −0.0 is considered to be less than the value +0.0. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

Note that in version 2.2 of the F extension, the FMIN.S and FMAX.S instructions were amended to implement the proposed IEEE 754-201x `minimumNumber` *and* `maximumNumber` *operations, rather than the IEEE 754-2008 minNum and maxNum operations. These operations differ in their handling of signaling NaNs.*



Floating-point fused multiply-add instructions require a new standard instruction format. R4-type instructions specify three source registers (*rs1*, *rs2*, and *rs3*) and a destination register (*rd*). This format is only used by the floating-point fused multiply-add instructions.

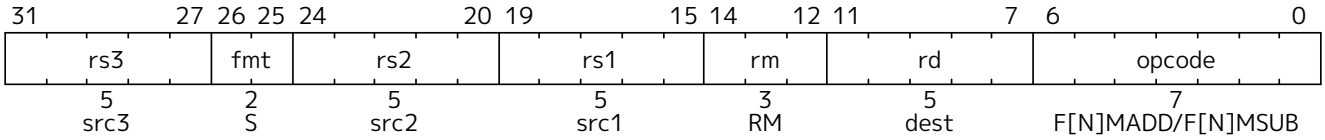
FMADD.S multiplies the values in *rs1* and *rs2*, adds the value in *rs3*, and writes the final result to *rd*. FMADD.S computes *(rs1×rs2)+rs3*.

FMSUB.S multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. FMSUB.S computes *(rs1×rs2)−rs3*.

FNMSUB.S multiplies the values in *rs1* and *rs2*, negates the product, adds the value in *rs3*, and writes the final result to *rd*. FNMSUB.S computes *−(rs1×rs2)+rs3*.

FNMADD.S multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. FNMADD.S computes $-(rs1 \times rs2) - rs3$.

The FNMSUB and FNMADD instructions are counterintuitively named, owing to the naming of the corresponding instructions in MIPS-IV. The MIPS instructions were defined to negate the sum, rather than negating the product as the RISC-V instructions do, so the naming scheme was more rational at the time. The two definitions differ with respect to signed-zero results. The RISC-V definition matches the behavior of the x86 and ARM fused multiply-add instructions, but unfortunately the RISC-V FNMSUB and FNMADD instruction names are swapped as compared to x86, whereas the RISC-V FMSUB and FNMSUB instruction names are swapped as compared to ARM.



The fused multiply-add (FMA) instructions consume a large part of the 32-bit instruction encoding space. Some alternatives considered were to restrict FMA to only use dynamic rounding modes, but static rounding modes are useful in code that exploits the lack of product rounding. Another alternative would have been to use rd to provide rs3, but this would require additional move instructions in some common sequences. The current design still leaves a large portion of the 32-bit encoding space open while avoiding having FMA be non-orthogonal.

The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

The IEEE 754-2008 standard permits, but does not require, raising the invalid exception for the operation ∞×0 + qNaN.

21.7. Single-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.S or FCVT.L.S converts a floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.S.W or FCVT.S.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a floating-point number in floating-point register *rd*. FCVT.WU.S, FCVT.LU.S, FCVT.S.WU, and FCVT.S.LU variants convert to or from unsigned integer values. For XLEN>32, FCVT.W[U].S sign-extends the 32-bit result to the destination register width. FCVT.L[U].S and FCVT.S.L[U] are RV64-only instructions. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. Table 27 gives the range of valid inputs for FCVT.*int*.S and the behavior for invalid inputs.

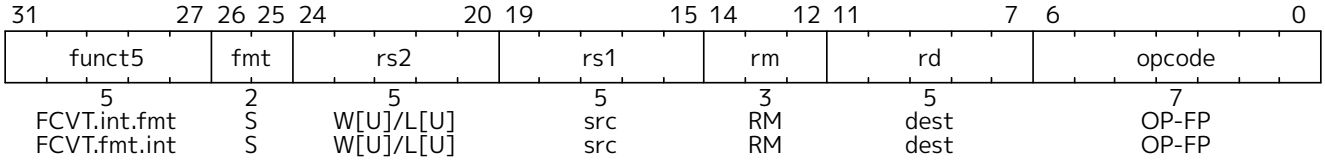
All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using FCVT.S.W *rd*, `x0`, which will never set any exception flags.

Table 27. Domains of float-to-integer conversions and behavior for invalid inputs

	FCVT.W.S	FCVT.WU.S	FCVT.L.S	FCVT.LU.S
Minimum valid input (after rounding)	-2^{31}	0	-2^{63}	0
Maximum valid input (after rounding)	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
Output for out-of-range negative input	-2^{31}	0	-2^{63}	0
Output for -∞	-2^{31}	0	-2^{63}	0

	FCVT.W.S	FCVT.WU.S	FCVT.L.S	FCVT.LU.S
Output for out-of-range positive input	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
Output for $+\infty$ or NaN	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.



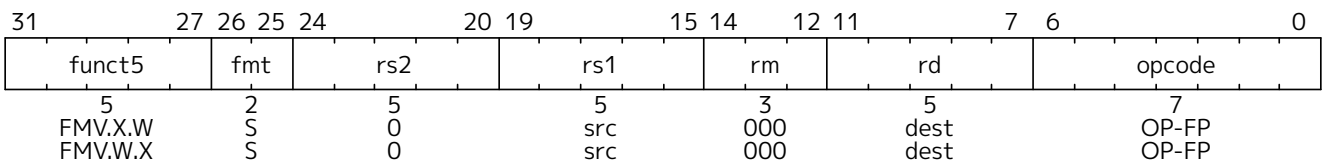
Floating-point to floating-point sign-injection instructions, FSGNJ.S, FSGNJN.S, and FSGNJX.S, produce a result that takes all bits except the sign bit from *rs1*. For FSGNJ, the result's sign bit is *rs2*'s sign bit; for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit; and for FSGNJX, the sign bit is the XOR of the sign bits of *rs1* and *rs2*. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs. Note, FSGNJ.S *rx, ry, ry* moves *ry* to *rx* (assembler pseudoinstruction FMV.S *rx, ry*); FSGNJN.S *rx, ry, ry* moves the negation of *ry* to *rx* (assembler pseudoinstruction FNEG.S *rx, ry*); and FSGNJX.S *rx, ry, ry* moves the absolute value of *ry* to *rx* (assembler pseudoinstruction FABS.S *rx, ry*).

The sign-injection instructions provide floating-point MV, ABS, and NEG, as well as supporting a few other operations, including the IEEE `copySign` *operation and sign manipulation in transcendental math function libraries. Although MV, ABS, and NEG only need a single register operand, whereas FSGNJ instructions need two, it is unlikely most microarchitectures would add optimizations to benefit from the reduced number of register reads for these relatively infrequent instructions. Even in this case, a microarchitecture can simply detect when both source registers are the same for FSGNJ instructions and only read a single copy.*

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.W moves the single-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the lower 32 bits of integer register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

FMV.W.X moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register *rs1* to the floating-point register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

The FMV.W.X and FMV.X.W instructions were previously called FMV.S.X and FMV.X.S. The use of W is more consistent with their semantics as an instruction that moves 32 bits without interpreting them. This became clearer after defining NaN-boxing. To avoid disturbing existing code, both the W and S versions will be supported by tools.

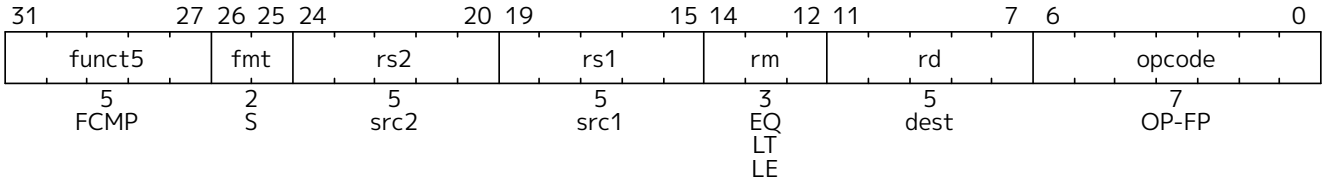


The base floating-point ISA was defined so as to allow implementations to employ an internal recoding of the floating-point format in registers to simplify handling of subnormal values and possibly to reduce functional unit latency. To this end, the F extension avoids representing integer values in the floating-point registers by defining conversion and comparison operations that read and write the integer register file directly. This also removes many of the common cases where explicit moves between integer and floating-point registers are required, reducing instruction count and critical paths for common mixed-format code sequences.

21.8. Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers (*rs1* = *rs2*, *rs1* < *rs2*, *rs1* ≤ *rs2*) writing 1 to the integer register *rd* if the condition holds, and 0 otherwise.

FLT.S and FLE.S perform what the IEEE 754-2008 standard refers to as *signaling* comparisons: that is, they set the invalid operation exception flag if either input is NaN. FEQ.S performs a *quiet* comparison: it only sets the invalid operation exception flag if either input is a signaling NaN. For all three instructions, the result is 0 if either operand is NaN.



The F extension provides a ≤ comparison, whereas the base ISAs provide a ≥ branch comparison. Because ≤ can be synthesized from ≥ and vice-versa, there is no performance implication to this inconsistency, but it is nevertheless an unfortunate incongruity in the ISA.

21.9. Single-Precision Floating-Point Classify Instruction

The FCLASS.S instruction examines the value in floating-point register *rs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in Table 28. The corresponding bit in *rd* will be set if the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in *rd* will be set. FCLASS.S does not set the floating-point exception flags.

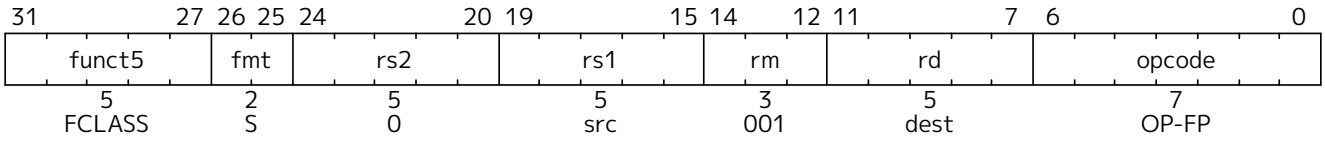


Table 28. Format of result of FCLASS instruction.

rd bit	Meaning
0	*rs1* is −∞.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is −0.
4	*rs1* is +0.
5	*rs1* is a positive subnormal number.

rd bit	Meaning
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

Chapter 22. "D" Extension for Double-Precision Floating-Point, Version 2.2

This chapter describes the standard double-precision floating-point instruction-set extension, which is named "D" and adds double-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The D extension depends on the base single-precision instruction subset F.

22.1. D Register State

The D extension widens the 32 floating-point registers, `f0-f31`, to 64 bits (FLEN=64 in Table 23. The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section 22.2.

FLEN can be 32, 64, or 128 depending on which of the F, D, and Q extensions are supported. There can be up to four different floating-point precisions supported, including H, F, D, and Q.

22.2. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower *n*-bit types, *n*<FLEN, are represented in the lower *n* bits of an FLEN-bit NaN value, in a process termed NaN-boxing. The upper bits of a valid NaN-boxed value must be all 1s. Valid NaN-boxed *n*-bit values therefore appear as negative quiet NaNs (qNaNs) when viewed as any wider *m*-bit value, $n < m \leq$ FLEN. Any operation that writes a narrower result to an 'f' register must write all 1s to the uppermost FLEN-*n* bits to yield a legal NaN-boxedvalue.

Software might not know the current type of data stored in a floating-point register but has to be able to save and restore the register values, hence the result of using wider operations to transfer narrower values has to be defined. A common case is for callee-saved registers, but a standard convention is also desirable for features including variadic functions, user-level threading libraries, virtual machine migration, and debugging.

Floating-point *n*-bit transfer operations move external values held in IEEE standard formats into and out of the `f` registers, and comprise floating-point loads and stores (FL*n*/FS*n*) and floating-point move instructions (FMV.*n*.X/FMV.X.*n*). A narrower *n*-bit transfer, *n*<FLEN, into the `f` registers will create a valid NaN-boxed value. A narrower *n*-bit transfer out of the floating-point registers will transfer the lower *n* bits of the register ignoring the upper FLEN-*n* bits.

Apart from transfer operations described in the previous paragraph, all other floating-point operations on narrower *n*-bit operations, *n*<FLEN, check if the input operands are correctly NaN-boxed, i.e., all upper FLEN-*n* bits are 1. If so, the *n* least-significant bits of the input are used as the input value, otherwise the input value is treated as an *n*-bit canonical NaN.

Earlier versions of this document did not define the behavior of feeding the results of narrower or wider operands into an operation, except to require that wider saves and restores would preserve the value of a narrower operand. The new definition removes this implementation-specific behavior, while still accommodating both non-recoded and recoded implementations of the floating-point unit. The new definition also helps catch software errors by propagating NaNs if values are used incorrectly.

Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a

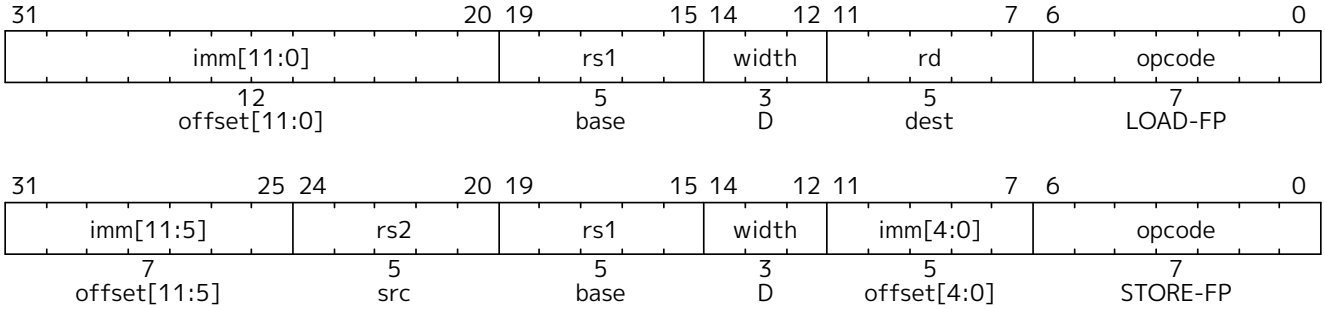
legal NaN-boxed value, and in writing all 1s to the upper bits of a result.

Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can be done without adding new state bits by recoding NaNs internally in the exponent field. Small modifications are needed to the pipelines used to transfer values in and out of the recoded format, but the datapath and latency costs are minimal. The recoding process has to handle shifting of input subnormal values for wide operands in any case, and extracting the NaN-boxed value is a similar process to normalization except for skipping over leading-1 bits instead of skipping over leading-0 bits, allowing the datapath multiplexing to be shared.

22.3. Double-Precision Load and Store Instructions

The FLD instruction loads a double-precision floating-point value from memory into floating-point register *rd*. FSD stores a double-precision value from the floating-point registers to memory.

The double-precision value may be a NaN-boxed single-precision value.

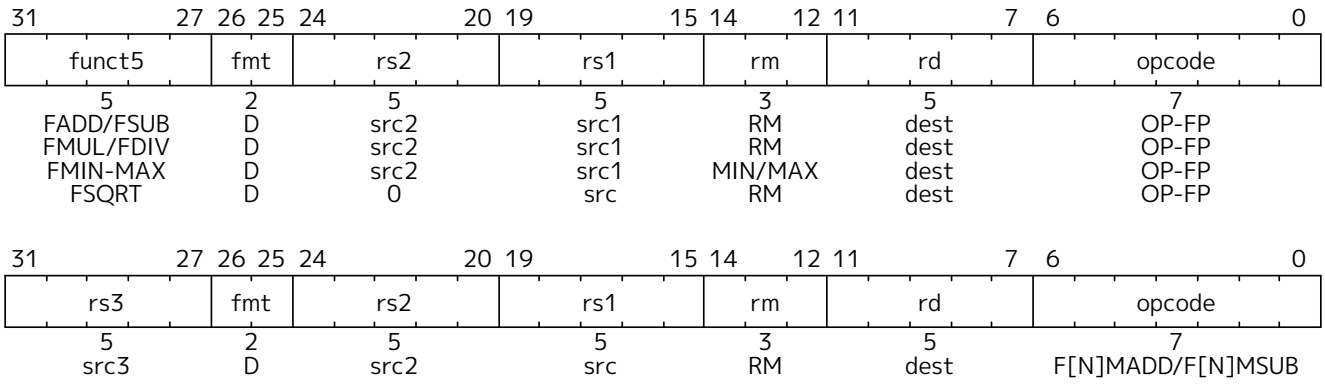


FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64.

FLD and FSD do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

22.4. Double-Precision Floating-Point Computational Instructions

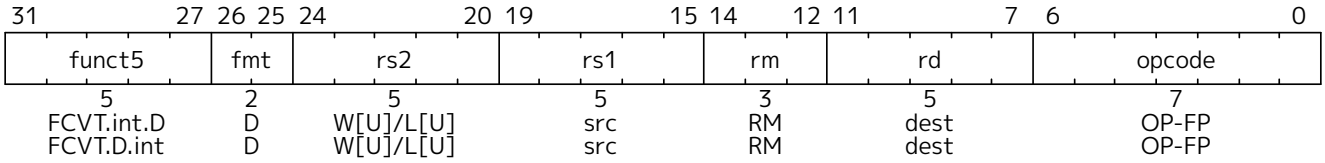
The double-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands and produce double-precision results.



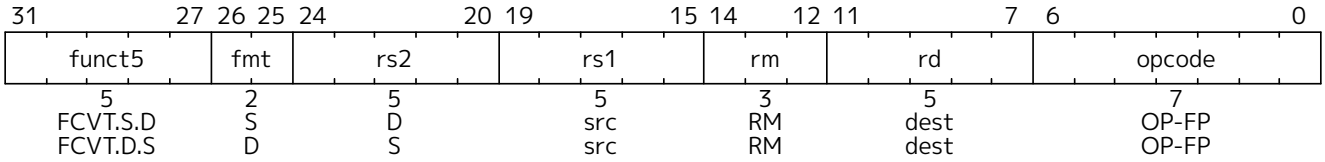
22.5. Double-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.D or FCVT.L.D converts a double-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.D.W or FCVT.D.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a double-precision floating-point number in floating-point register *rd*. FCVT.WU.D, FCVT.LU.D, FCVT.D.WU, and FCVT.D.LU variants convert to or from unsigned integer values. For RV64, FCVT.W[U].D sign-extends the 32-bit result. FCVT.L[U].D and FCVT.D.L[U] are RV64-only instructions. The range of valid inputs for FCVT.*int*.D and the behavior for invalid inputs are the same as for FCVT.*int*.S.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. Note FCVT.D.W[U] always produces an exact result and is unaffected by rounding mode.



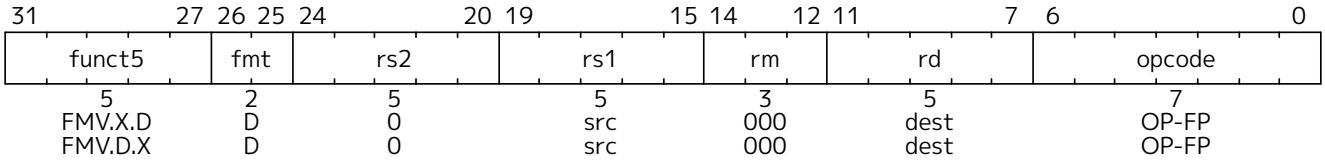
The double-precision to single-precision and single-precision to double-precision conversion instructions, FCVT.S.D and FCVT.D.S, are encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The *rs2* field encodes the datatype of the source, and the *fmt* field encodes the datatype of the destination. FCVT.S.D rounds according to the RM field; FCVT.D.S will never round.



Floating-point to floating-point sign-injection instructions, FSGNJ.D, FSGNJN.D, and FSGNJX.D are defined analogously to the single-precision sign-injection instruction.

For XLEN≥64 only, instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.D moves the double-precision value in floating-point register *rs1* to a representation in IEEE 754-2008 standard encoding in integer register *rd*. FMV.D.X moves the double-precision value encoded in IEEE 754-2008 standard encoding from the integer register *rs1* to the floating-point register *rd*.

FMV.X.D and FMV.D.X do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.



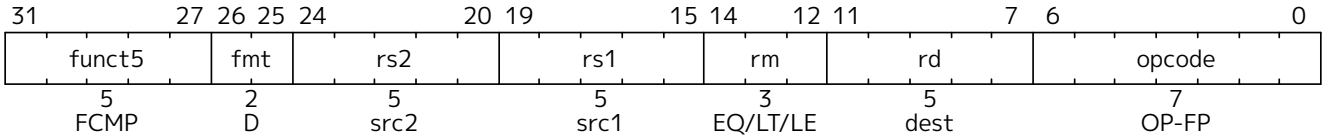
Early versions of the RISC-V ISA had additional instructions to allow RV32 systems to transfer between the upper and lower portions of a 64-bit floating-point register and an integer register. However, these would be the only instructions with partial register writes and would

add complexity in implementations with recoded floating-point or register renaming, requiring a pipeline read-modify-write sequence. Scaling up to handling quad-precision for RV32 and RV64 would also require additional instructions if they were to follow this pattern. The ISA was defined to reduce the number of explicit int-float register moves, by having conversions and comparisons write results to the appropriate register file, so we expect the benefit of these instructions to be lower than for other ISAs.

We note that for systems that implement a 64-bit floating-point unit including fused multiply-add support and 64-bit floating-point loads and stores, the marginal hardware cost of moving from a 32-bit to a 64-bit integer datapath is low, and a software ABI supporting 32-bit wide address-space and pointers can be used to avoid growth of static data and dynamic memory traffic.

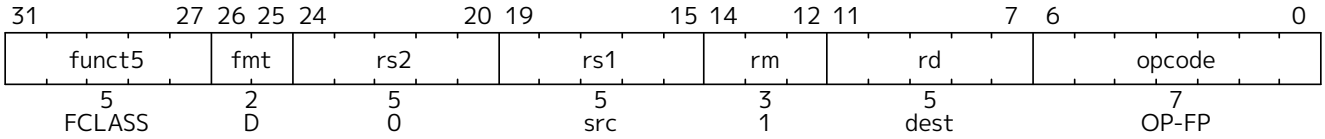
22.6. Double-Precision Floating-Point Compare Instructions

The double-precision floating-point compare instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands.



22.7. Double-Precision Floating-Point Classify Instruction

The double-precision floating-point classify instruction, FCLASS.D, is defined analogously to its single-precision counterpart, but operates on double-precision operands.

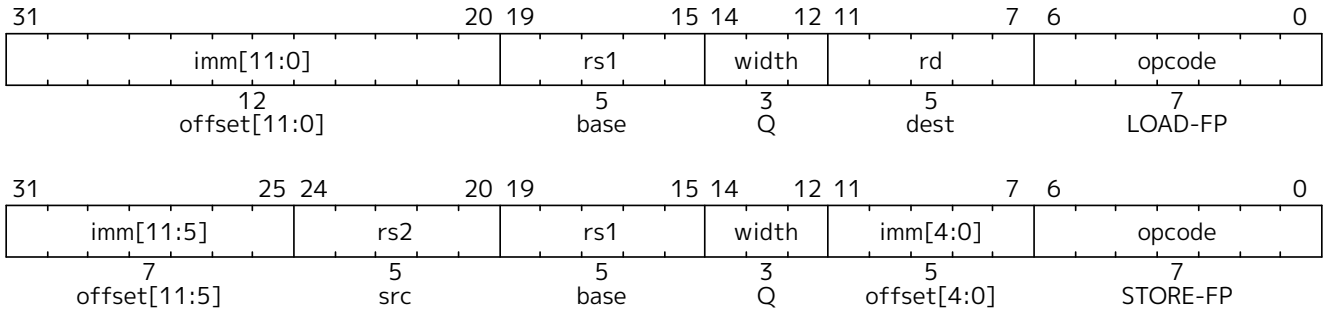


Chapter 23. "Q" Extension for Quad-Precision Floating-Point, Version 2.2

This chapter describes the Q standard extension for 128-bit quad-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The quad-precision binary floating-point instruction-set extension is named "Q"; it depends on the double-precision floating-point extension D. The floating-point registers are now extended to hold either a single, double, or quad-precision floating-point value (FLEN=128). The NaN-boxing scheme described in Section 22.2 is now extended recursively to allow a single-precision value to be NaN-boxed inside a double-precision value which is itself NaN-boxed inside a quad-precision value.

23.1. Quad-Precision Load and Store Instructions

New 128-bit variants of LOAD-FP and STORE-FP instructions are added, encoded with a new value for the funct3 width field.



FLQ and FSQ are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN=128.

FLQ and FSQ do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

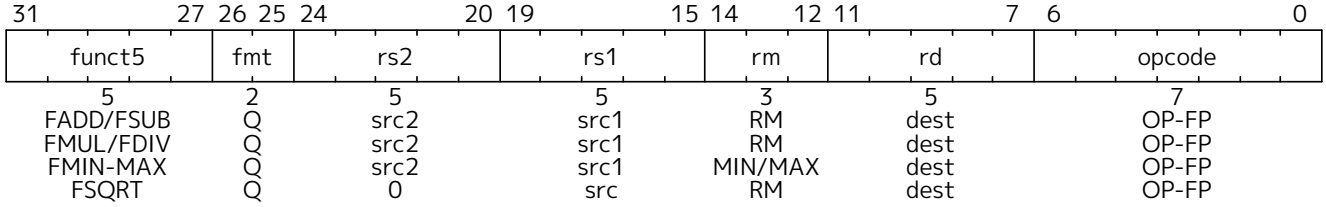
23.2. Quad-Precision Computational Instructions

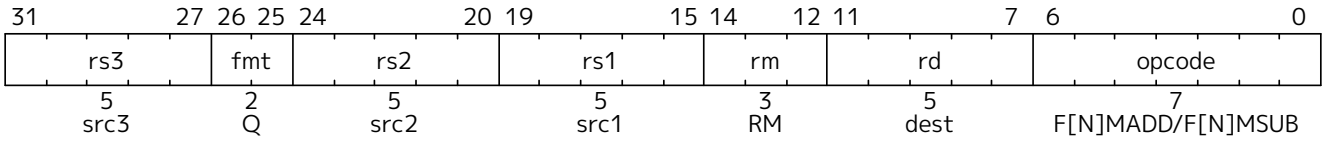
A new supported format is added to the format field of most instructions, as shown in Table 29

Table 29. Format field encoding.

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

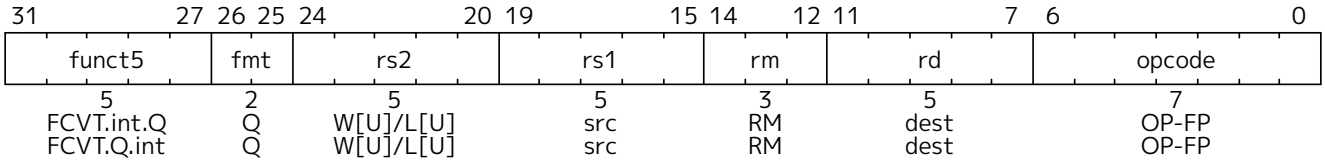
The quad-precision floating-point computational instructions are defined analogously to their double-precision counterparts, but operate on quad-precision operands and produce quad-precision results.



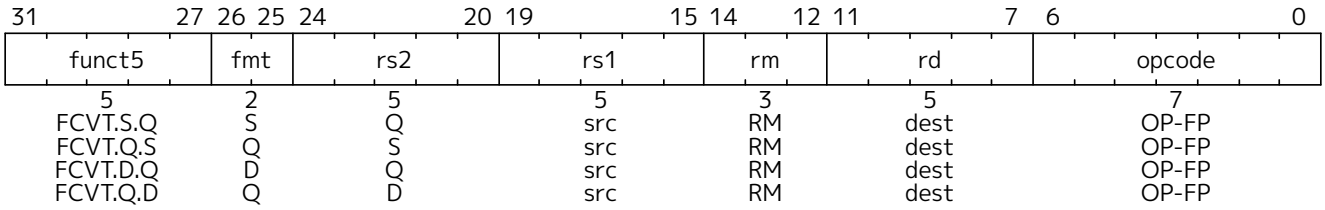


23.3. Quad-Precision Convert and Move Instructions

New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision-to-integer and integer-to-double-precision conversion instructions. FCVT.W.Q or FCVT.L.Q converts a quad-precision floating-point number to a signed 32-bit or 64-bit integer, respectively. FCVT.Q.W or FCVT.Q.L converts a 32-bit or 64-bit signed integer, respectively, into a quad-precision floating-point number. FCVT.WU.Q, FCVT.LU.Q, FCVT.Q.WU, and FCVT.Q.LU variants convert to or from unsigned integer values. FCVT.L[U].Q and FCVT.Q.L[U] are RV64-only instructions. Note FCVT.Q.L[U] always produces an exact result and is unaffected by rounding mode.



New floating-point-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision floating-point-to-floating-point conversion instructions. FCVT.S.Q or FCVT.Q.S converts a quad-precision floating-point number to a single-precision floating-point number, or vice-versa, respectively. FCVT.D.Q or FCVT.Q.D converts a quad-precision floating-point number to a double-precision floating-point number, or vice-versa, respectively.



Floating-point to floating-point sign-injection instructions, FSGNJ.Q, FSGNJN.Q, and FSGNJX.Q are defined analogously to the double-precision sign-injection instruction.

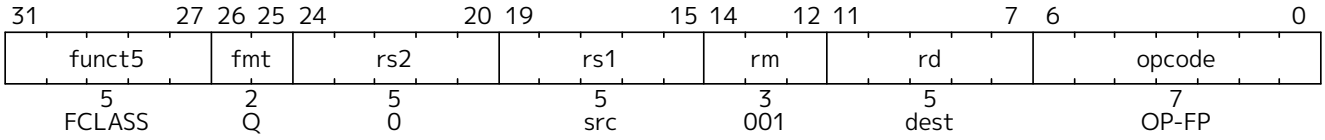
FMV.X.Q and FMV.Q.X instructions are not provided in RV32 or RV64, so quad-precision bit patterns must be moved to the integer registers via memory.

23.4. Quad-Precision Floating-Point Compare Instructions

The quad-precision floating-point compare instructions are defined analogously to their double-precision counterparts, but operate on quad-precision operands.

23.5. Quad-Precision Floating-Point Classify Instruction

The quad-precision floating-point classify instruction, FCLASS.Q, is defined analogously to its double-precision counterpart, but operates on quad-precision operands.



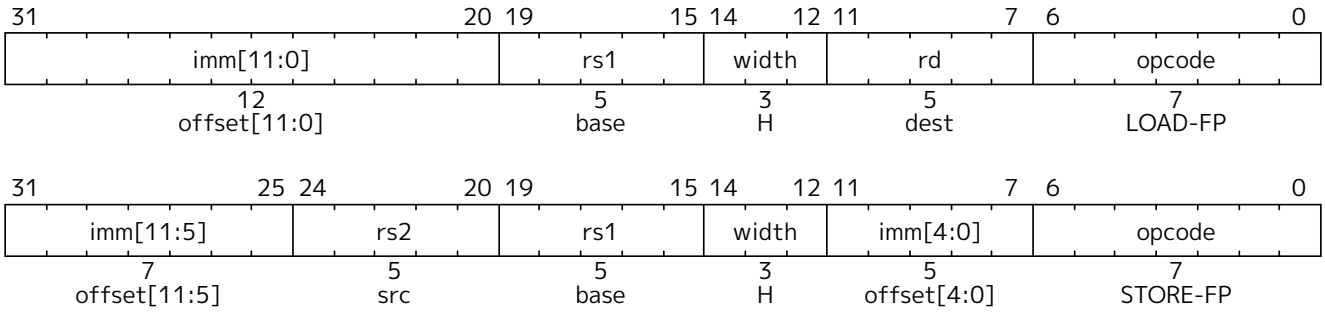
Chapter 24. "Zfh" and "Zfhmin" Extensions for Half-Precision Floating-Point, Version 1.0

This chapter describes the Zfh standard extension for 16-bit half-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The Zfh extension depends on the single-precision floating-point extension, F. The NaN-boxing scheme described in Section 22.2 is extended to allow a half-precision value to be NaN-boxed inside a single-precision value (which may be recursively NaN-boxed inside a double- or quad-precision value when the D or Q extension is present).

This extension primarily provides instructions that consume half-precision operands and produce half-precision results. However, it is also common to compute on half-precision data using higher intermediate precision. Although this extension provides explicit conversion instructions that suffice to implement that pattern, future extensions might further accelerate such computation with additional instructions that implicitly widen their operands—e.g., half×half+single→single—or implicitly narrow their results—e.g., half+single→half.

24.1. Half-Precision Load and Store Instructions

New 16-bit variants of LOAD-FP and STORE-FP instructions are added, encoded with a new value for the funct3 width field.



FLH and FSH are only guaranteed to execute atomically if the effective address is naturally aligned.

FLH and FSH do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved. FLH NaN-boxes the result written to *rd*, whereas FSH ignores all but the lower 16 bits in *rs2*.

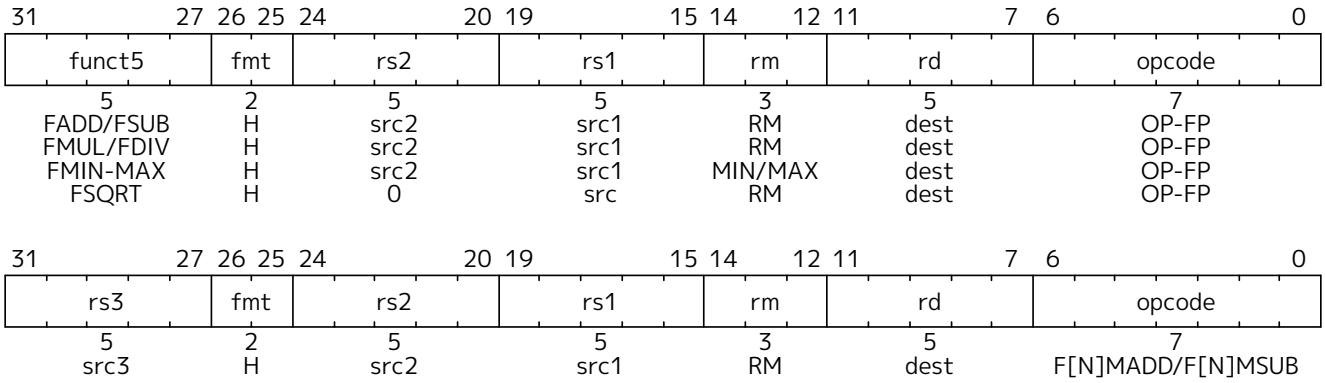
24.2. Half-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 30.

Table 30. Format field encoding.

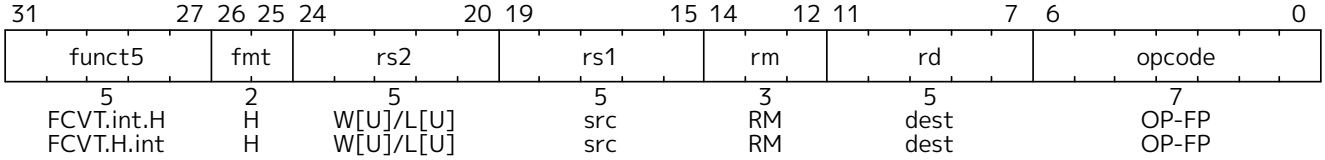
fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

The half-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on half-precision operands and produce half-precision results.

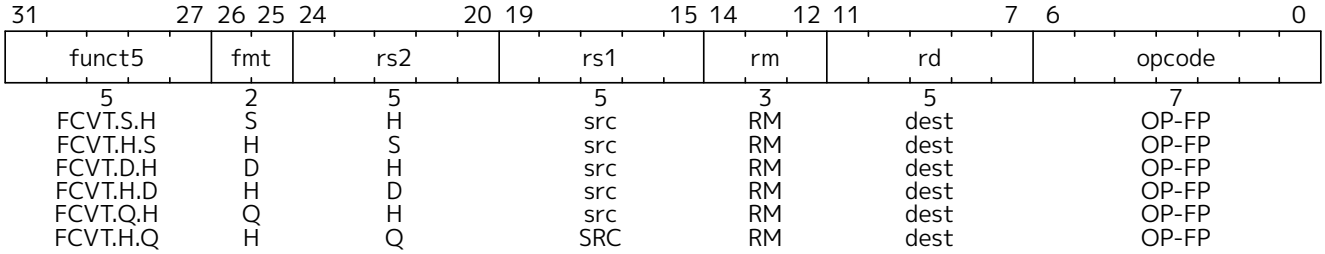


24.3. Half-Precision Conversion and Move Instructions

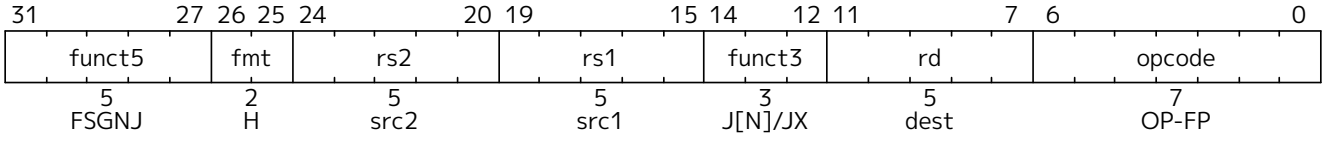
New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the single-precision-to-integer and integer-to-single-precision conversion instructions. FCVT.W.H or FCVT.L.H converts a half-precision floating-point number to a signed 32-bit or 64-bit integer, respectively. FCVT.H.W or FCVT.H.L converts a 32-bit or 64-bit signed integer, respectively, into a half-precision floating-point number. FCVT.WU.H, FCVT.LU.H, FCVT.H.WU, and FCVT.H.LU variants convert to or from unsigned integer values. FCVT.L[U].H and FCVT.H.L[U] are RV64-only instructions.



New floating-point-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision floating-point-to-floating-point conversion instructions. FCVT.S.H or FCVT.H.S converts a half-precision floating-point number to a single-precision floating-point number, or vice-versa, respectively. If the D extension is present, FCVT.D.H or FCVT.H.D converts a half-precision floating-point number to a double-precision floating-point number, or vice-versa, respectively. If the Q extension is present, FCVT.Q.H or FCVT.H.Q converts a half-precision floating-point number to a quad-precision floating-point number, or vice-versa, respectively.



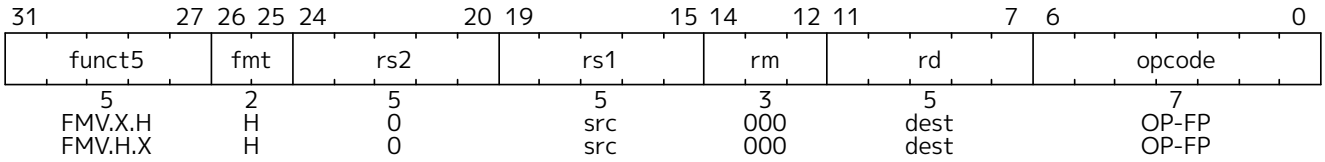
Floating-point to floating-point sign-injection instructions, FSGNJ.H, FSGNJN.H, and FSGNJX.H are defined analogously to the single-precision sign-injection instruction.



Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.H moves the half-precision value in floating-point register *rs1* to a representation in IEEE 754-2008 standard encoding in integer register *rd*, filling the upper XLEN-16 bits with copies of the floating-point number's sign bit.

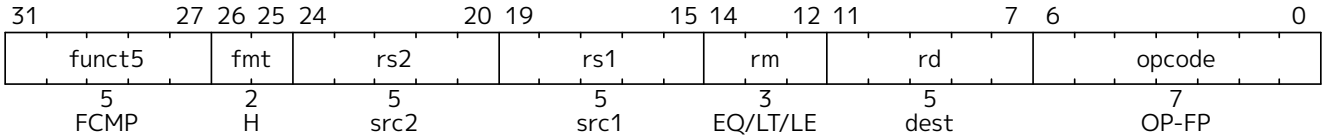
FMV.H.X moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register *rs1* to the floating-point register *rd*, NaN-boxing the result.

FMV.X.H and FMV.H.X do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.



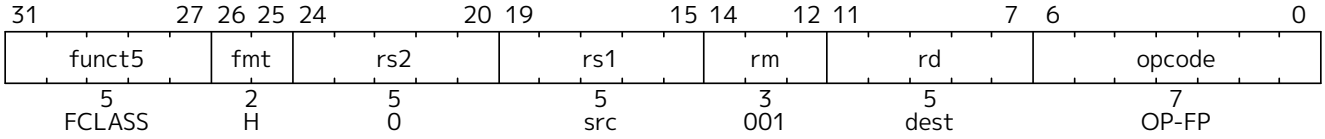
24.4. Half-Precision Floating-Point Compare Instructions

The half-precision floating-point compare instructions are defined analogously to their single-precision counterparts, but operate on half-precision operands.



24.5. Half-Precision Floating-Point Classify Instruction

The half-precision floating-point classify instruction, FCLASS.H, is defined analogously to its single-precision counterpart, but operates on half-precision operands.



24.6. "Zfhmin" Standard Extension for Minimal Half-Precision Floating-Point

This section describes the Zfhmin standard extension, which provides minimal support for 16-bit half-precision binary floating-point instructions. The Zfhmin extension is a subset of the Zfh extension, consisting only of data transfer and conversion instructions. Like Zfh, the Zfhmin extension depends on the single-precision floating-point extension, F. The expectation is that Zfhmin software primarily uses the half-precision format for storage, performing most computation in higher precision.

The Zfhmin extension includes the following instructions from the Zfh extension: FLH, FSH, FMV.X.H, FMV.H.X, FCVT.S.H, and FCVT.H.S. If the D extension is present, the FCVT.D.H and FCVT.H.D instructions are also included. If the Q extension is present, the FCVT.Q.H and FCVT.H.Q instructions are additionally included.

Zfhmin does not include the FSGNJ.H instruction, because it suffices to instead use the FSGNJ.S instruction to move half-precision values between floating-point registers.

Half-precision addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the half-precision operands to single-precision, performing the operation using single-precision arithmetic, then converting back to half-precision. (Roux, 2014) Performing half-precision fused multiply-addition using this method incurs a 1-ulp error on some inputs for the RNE and RMM rounding modes.

Conversion from 8- or 16-bit integers to half-precision can be emulated by first converting to single-precision, then converting to half-precision. Conversion from 32-bit integer can be emulated by first converting to double-precision. If the D extension is not present and a 1-ulp error under RNE or RMM is tolerable, 32-bit integers can be first converted to single-precision instead. The same remark applies to conversions from 64-bit integers without the Q extension.

Chapter 25. "BF16" Extensions for BFloat16-precision Floating-Point, Version 1.0

25.1. Introduction

When FP16 (officially called binary16) was first introduced by the IEEE-754 standard, it was just an interchange format. It was intended as a space/bandwidth efficient encoding that would be used to transfer information. This is in line with the Zfhmin extension.

However, there were some applications (notably graphics) that found that the smaller precision and dynamic range was sufficient for their space. So, FP16 started to see some widespread adoption as an arithmetic format. This is in line with the Zfh extension.

While it was not the intention of '754 to have FP16 be an arithmetic format, it is supported by the standard. Even though the '754 committee recognized that FP16 was gaining popularity, the committee decided to hold off on making it a basic format in the 2019 release. This means that a '754 compliant implementation of binary floating point, which needs to support at least one basic format, cannot support only FP16 - it needs to support at least one of binary32, binary64, and binary128.

Experts working in machine learning noticed that FP16 was a much more compact way of storing operands and often provided sufficient precision for them. However, they also found that intermediate values were much better when accumulated into a higher precision. The final computations were then typically converted back into the more compact FP16 encoding. This approach has become very common in machine learning (ML) inference where the weights and activations are stored in FP16 encodings. There was the added benefit that smaller multiplication blocks could be created for the FP16's smaller number of significant bits. At this point, widening multiply-accumulate instructions became much more common. Also, more complicated dot product instructions started to show up including those that packed two FP16 numbers in a 32-bit register, multiplied these by another pair of FP16 numbers in another register, added these two products to an FP32 accumulate value in a 3rd register and returned an FP32 result.

Experts working in machine learning at Google who continued to work with FP32 values noted that the least significant 16 bits of their mantissas were not always needed for good results, even in training. They proposed a truncated version of FP32, which was the 16 most significant bits of the FP32 encoding. This format was named BFloat16 (or BF16). The B in BF16, stands for Brain since it was initially introduced by the Google Brain team. Not only did they find that the number of significant bits in BF16 tended to be sufficient for their work (despite being fewer than in FP16), but it was very easy for them to reuse their existing data; FP32 numbers could be readily rounded to BF16 with a minimal amount of work. Furthermore, the even smaller number of the BF16 significant bits enabled even smaller multiplication blocks to be built. Similar to FP16, BF16 multiply-accumulate widening and dot-product instructions started to proliferate.

25.2. Intended Audience

Floating-point arithmetic is a specialized subject, requiring people with many different backgrounds to cooperate in its correct and efficient implementation. Where possible, we have written this specification to be understandable by all, though we recognize that the motivations and references to algorithms or other specifications and standards may be unfamiliar to those who are not domain experts.

This specification anticipates being read and acted on by various people with different backgrounds. We have tried to capture these backgrounds here, with a brief explanation of what we expect them to know, and how it relates to the specification. We hope this aids people's understanding of which aspects of the specification are particularly relevant to them, and which they may (safely!) ignore or pass to a colleague.

Software developers

These are the people we expect to write code using the instructions in this specification. They should understand the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards to which we refer.

Computer architects

We expect architects to have some basic floating-point background. Furthermore, we expect architects to be able to examine our instructions for implementation issues, understand how the instructions will be used in context, and advise on how they best to fit the functionality.

Digital design engineers & micro-architects

These are the people who will implement the specification inside a core. Floating-point expertise is assumed as not all of the corner cases are pointed out in the specification.

Verification engineers

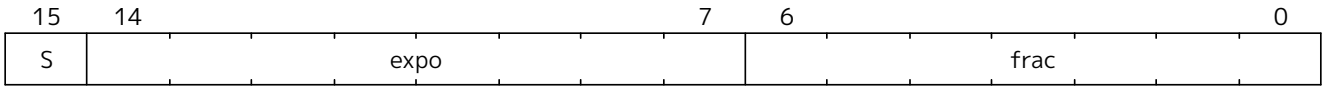
Responsible for ensuring the correct implementation of the extension in hardware. These people are expected to have some floating-point expertise so that they can identify and generate the interesting corner cases --- include exceptions --- that are common in floating-point architectures and implementations.

These are by no means the only people concerned with the specification, but they are the ones we considered most while writing it.

25.3. Number Format

25.3.1. BF16 Operand Format

BF16 bits



IEEE Compliance: While BF16 (also known as BFloat16) is not an IEEE-754 *standard* format, it is a valid floating-point format as defined by IEEE-754. There are three parameters that specify a format: radix (b), number of digits in the significand (p), and maximum exponent (emax). For BF16 these values are:

Table 31. BF16 parameters

Parameter	Value
radix (b)	2
significand (p)	8
emax	127

Table 32. Obligatory Floating Point Format Table

Format	Sign Bits	Expo Bits	fraction bits	padded 0s	encoding bits	expo max/bias	expo min
FP16	1	5	10	0	16	15	-14
BF16	1	8	7	0	16	127	-126
TF32	1	8	10	13	32	127	-126
FP32	1	8	23	0	32	127	-126

Format	Sign Bits	Expo Bits	fraction bits	padded 0s	encoding bits	expo max/bias	expo min
FP64	1	11	52	0	64	1023	-1022
FP128	1	15	112	0	128	16,383	-16,382

25.3.2. BF16 Behavior

For these BF16 extensions, instruction behavior on BF16 operands is the same as for other floating-point instructions in the RISC-V ISA. For easy reference, some of this behavior is repeated here.

25.3.2.1. Subnormal Numbers:

Floating-point values that are too small to be represented as normal numbers, but can still be expressed by the format's smallest exponent value with a "0" integer bit and at least one "1" bit in the trailing fractional bits are called subnormal numbers. Basically, the idea is there is a trade off of precision to support *gradual underflow*.

All of the BF16 instructions in the extensions defined in this specification (i.e., Zfbfmin, Zvfbfmin, and Zvfbfwma) fully support subnormal numbers. That is, instructions are able to accept subnormal values as inputs and they can produce subnormal results.

Future floating-point extensions, including those that operate on BF16 values, may chose not to support subnormal numbers. The comments about supporting subnormal BF16 values are limited to those instructions defined in this specification.

25.3.2.2. Infinities:

Infinities are used to represent values that are too large to be represented by the target format. These are usually produced as a result of overflows (depending on the rounding mode), but can also be provided as inputs. Infinities have a sign associated with them: there are positive infinities and negative infinities.

Infinities are important for keeping meaningless results from being operated upon.

25.3.2.3. NaNs

NaN stands for Not a Number.

There are two types of NaNs: signalling (sNaN) and quiet (qNaN). No computational instruction will ever produce an sNaN; These are only provided as input data. Operating on an sNaN will cause an invalid operation exception. Operating on a Quiet NaN usually does not cause an exception.

QNaNs are provided as the result of an operation when it cannot be represented as a number or infinity. For example, performing the square root of -1 will result in a qNaN because there is no real number that can represent the result. NaNs can also be used as inputs.

NaNs include a sign bit, but the bit has no meaning.

NaNs are important for keeping meaningless results from being operated upon.

Except where otherwise explicitly stated, when the result of a floating-point operation is a qNaN, it is the RISC-V canonical NaN. For BF16, the RISC-V canonical NaN corresponds to the pattern of *0x7fc0* which is the most significant 16 bits of the RISC-V single-precision canonical NaN.

25.3.2.4. Scalar NaN Boxing

RISC-V applies NaN boxing to scalar results and checks for NaN boxing when a floating-point operation --- even a vector-scalar operation --- consumes a value from a scalar floating-point register. If the value is properly NaN-boxed, its least significant bits are used as the operand, otherwise it is treated as if it were the canonical QNaN.

NaN boxing is nothing more than putting the smaller encoding in the least significant bits of a register and setting all of the more significant bits to "1". This matches the encoding of a qNaN (although not the canonical NaN) in the larger precision.

Nan-boxing never affects the value of the operand itself, it just changes the bits of the register that are more significant than the operand's most significant bit.

25.3.2.5. Rounding Modes:

As is the case with other floating-point instructions, the BF16 instructions support all 5 RISC-V Floating-point rounding modes. These modes can be specified in the `rm` field of scalar instructions as well as in the `frm` CSR

Table 33. RISC-V Floating Point Rounding Modes

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude

As with other scalar floating-point instructions, the rounding mode field `rm` can also take on the `DYN` encoding, which indicates that the instruction uses the rounding mode specified in the `frm` CSR.

Table 34. Additional encoding for the `rm` field of scalar instructions

Rounding Mode	Mnemonic	Meaning
111	DYN	select dynamic rounding mode

In practice, the default IEEE rounding mode (round to nearest, ties to even) is generally used for arithmetic.

25.3.2.6. Handling exceptions

RISC-V supports IEEE-defined default exception handling. BF16 is no exception.

Default exception handling, as defined by IEEE, is a simple and effective approach to producing results in exceptional cases. For the coder to be able to see what has happened, and take further action if needed, BF16 instructions set floating-point exception flags the same way as all other floating-point instructions in RISC-V.

25.3.2.6.1. Underflow

The IEEE-defined underflow exception requires that a result be inexact and tiny, where tininess can be detected before or after rounding. In RISC-V, tininess is detected after rounding.

It is important to note that the detection of tininess after rounding requires its own rounding that is different from the final result rounding. This tininess detection requires rounding as if the exponent were unbounded. This means that the input to the rounder is always a normal number. This is different from the final result rounding where the input to the rounder is a subnormal number when the value is too small to be represented as a normal number in the target format. The two different roundings can result in underflow being signalled for results that are rounded back to the normal range.

As is defined in '754, under default exception handling, underflow is only signalled when the result is tiny and inexact. In such a case, both the underflow and inexact flags are raised.

25.4. Extensions

The group of extensions introduced by the BF16 Instruction Set Extensions is listed here.

Detection of individual BF16 extensions uses the unified software-based RISC-V discovery method.

At the time of writing, these discovery mechanisms are still a work in progress.

The BF16 extensions defined in this specification (i.e., `Zfbfmin`, `Zvfbfmin`, and `Zvfbfwma`) depend on the single-precision floating-point extension `F`. Furthermore, the vector BF16 extensions (i.e.,`Zvfbfmin`, and `Zvfbfwma`) depend on the `"V"` Vector Extension for Application Processors or the `Zve32f` Vector Extension for Embedded Processors.

As stated later in this specification, there exists a dependency between the newly defined extensions: `Zvfbfwma` depends on `Zfbfmin` and `Zvfbfmin`.

This initial set of BF16 extensions provides very basic functionality including scalar and vector conversion between BF16 and single-precision values, and vector widening multiply-accumulate instructions.

25.4.1. `Zfbfmin` - Scalar BF16 Converts

This extension provides the minimal set of instructions needed to enable scalar support of the BF16 format. It enables BF16 as an interchange format as it provides conversion between BF16 values and FP32 values.

This extension depends upon the single-precision floating-point extension `F`.

This extension includes six instructions: the `FCVT.BF16.S` and `FCVT.S.BF16` instructions, defined below, and the `FLH`, `FSH`, `FMV.X.H`, and `FMV.H.X` instructions, defined in Chapter 24.

While conversion instructions tend to include all supported formats, in these extensions we only support conversion between BF16 and FP32 as we are targeting a special use case. These extensions are intended to support the case where BF16 values are used as reduced precision versions of FP32 values, where use of BF16 provides a two-fold advantage for storage, bandwidth, and computation. In this use case, the BF16 values are typically multiplied by each other and accumulated into FP32 sums. These sums are typically converted to BF16 and then used as subsequent inputs. The operations on the BF16 values can be performed on the CPU or a loosely coupled coprocessor.

Subsequent extensions might provide support for native BF16 arithmetic. Such extensions could add additional conversion instructions to allow all supported formats to be converted to and from BF16.

BF16 addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the BF16 operands to single-precision, performing the operation using single-precision arithmetic, and then converting back to BF16. Performing BF16 fused multiply-addition using this method can produce results that differ by 1-ulp on some inputs for the RNE and RMM rounding modes.

Conversions between BF16 and formats larger than FP32 can be emulated. Exact widening conversions from BF16 can be synthesized by first converting to FP32 and then converting from FP32 to the target precision. Conversions narrowing to BF16 can be synthesized by first converting to FP32 through a series of halving steps and then converting from FP32 to BF16. As with the fused multiply-addition instruction described above, this method of converting values to BF16 can be off by 1-ulp on some inputs for the RNE and RMM rounding modes.

Mnemonic	Instruction
FCVT.BF16.S	Convert FP32 to BF16
FCVT.S.BF16	Convert BF16 to FP32
FLH	
FSH	
FMV.H.X	
FMV.X.H	

25.4.2. `Zvfbfmin` - Vector BF16 Converts

This extension provides the minimal set of instructions needed to enable vector support of the BF16 format. It enables BF16 as an interchange format as it provides conversion between BF16 values and FP32 values.

This extension depends upon `Zve32f` vector extension.

While conversion instructions tend to include all supported formats, in these extensions we only support conversion between BF16 and FP32 as we are targeting a special use case. These extensions are intended to support the case where BF16 values are used as reduced precision versions of FP32 values, where use of BF16 provides a two-fold advantage for storage, bandwidth, and computation. In this use case, the BF16 values are typically multiplied by each other and accumulated into FP32 sums. These sums are typically converted to BF16 and then used as subsequent inputs. The operations on the BF16 values can be performed on the CPU or a loosely coupled coprocessor.

Subsequent extensions might provide support for native BF16 arithmetic. Such extensions could add additional conversion instructions to allow all supported formats to be converted to and from BF16.

BF16 addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the BF16 operands to single-precision, performing the operation using single-precision arithmetic, and then converting back to BF16. Performing BF16 fused multiply-addition using this method can produce results that differ by 1-ulp on some inputs for the RNE and RMM rounding modes.

Conversions between BF16 and formats larger than FP32 can be faithfully emulated. Exact widening conversions from BF16 can be synthesized by first converting to FP32 and then converting from FP32 to the target precision. Conversions narrowing to BF16 can be synthesized by first converting to FP32 through a series of halving steps using vector round-towards-odd narrowing conversion instructions (vfncvt.rod.f.f.w). The final convert from FP32 to BF16 would use the desired rounding mode.

Mnemonic	Instruction
vfncvtbf16.f.f.w	Vector convert FP32 to BF16
vfwcvtbf16.f.f.v	Vector convert BF16 to FP32

25.4.3. `Zvfbfwma` - Vector BF16 widening mul-add

This extension provides a vector widening BF16 mul-add instruction that accumulates into FP32.

This extension depends upon the `Zvfbfmin` extension and the `Zfbfmin` extension.

Mnemonic	Instruction
VFWMACCBF16	Vector BF16 widening multiply-accumulate

25.5. Instructions

25.5.1. fcvt.bf16.s

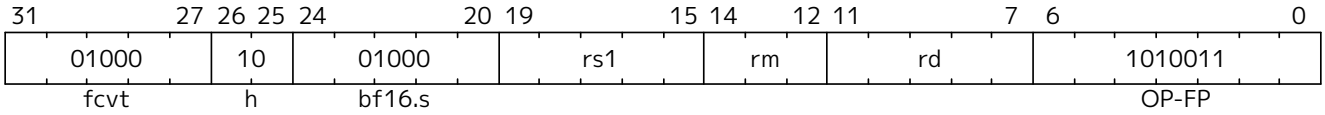
Synopsis

Convert FP32 value to a BF16 value

Mnemonic

fcvt.bf16.s rd, rs1

Encoding



Encoding

While the mnemonic of this instruction is consistent with that of the other RISC-V floating-point convert instructions, a new encoding is used in bits 24:20.

`BF16.S` *and* `H` *are used to signify that the source is FP32 and the destination is BF16.*

Description

Narrowing convert FP32 value to a BF16 value. Round according to the RM field.

This instruction is similar to other narrowing floating-point-to-floating-point conversion instructions.

Exceptions: Overflow, Underflow, Inexact, Invalid

Included in: Zfbfmin

25.5.2. fcvt.s.bf16

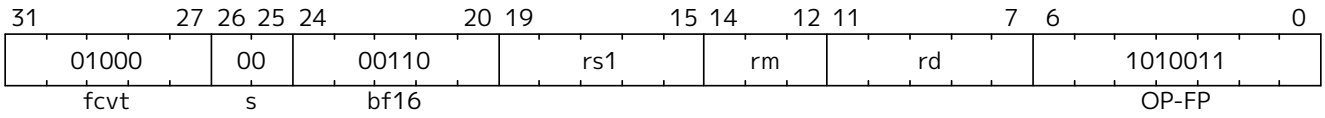
Synopsis

Convert BF16 value to an FP32 value

Mnemonic

fcvt.s.bf16 rd, rs1

Encoding



Encoding

While the mnemonic of this instruction is consistent with that of the other RISC-V floating-point convert instructions, a new encoding is used in bits 24:20 to indicate that the source is BF16.

Description

Converts a BF16 value to an FP32 value. The conversion is exact.

This instruction is similar to other widening floating-point-to-floating-point conversion instructions.

If the input is normal or infinity, the BF16 encoded value is shifted to the left by 16 places and the least significant 16 bits are written with 0s.

The result is NaN-boxed by writing the most significant `FLEN`*-32 bits with 1s.*

Exceptions: Invalid

Included in: Zfbfmin

25.5.3. vfncvtbf16.f.f.w

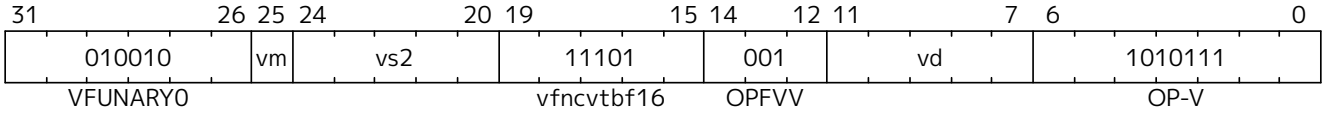
Synopsis

Vector convert FP32 to BF16

Mnemonic

vfncvtbf16.f.f.w vd, vs2, vm

Encoding



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vs2	input	32	FP32 Source
Vd	output	16	BF16 Result

Description

Narrowing convert from FP32 to BF16. Round according to the *frm* register.

This instruction is similar to `vfncvt.f.f.w` which converts a floating-point value in a 2*SEW-width format into an SEW-width format. However, here the SEW-width format is limited to BF16.

Exceptions: Overflow, Underflow, Inexact, Invalid

Included in: Zvfbfmin

25.5.4. vfwcvtbf16.f.f.v

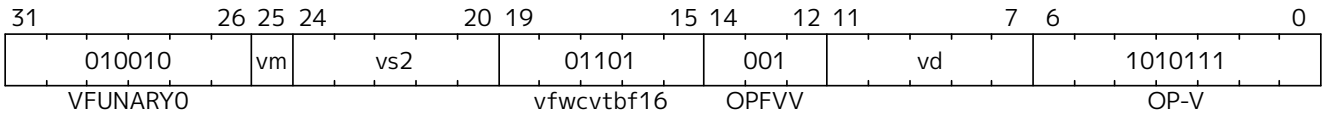
Synopsis

Vector convert BF16 to FP32

Mnemonic

vfwcvtbf16.f.f.v vd, vs2, vm

Encoding



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vs2	input	16	BF16 Source
Vd	output	32	FP32 Result

Description

Widening convert from BF16 to FP32. The conversion is exact.

This instruction is similar to `vfwcvt.f.f.v` which converts a floating-point value in an SEW-width format into a 2*SEW-width format. However, here the SEW-width format is limited to BF16.

If the input is normal or infinity, the BF16 encoded value is shifted to the left by 16 places and the least significant 16 bits are written with 0s.

Exceptions: Invalid

Included in: Zvfbfmin

25.5.5. vfwmaccbf16

Synopsis

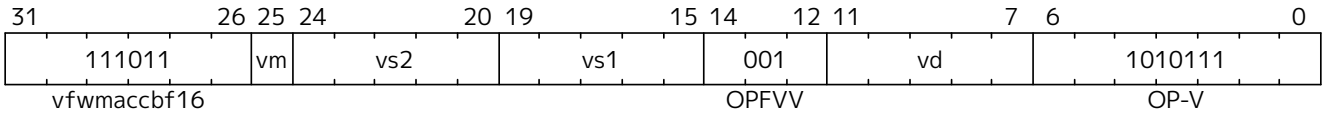
Vector BF16 widening multiply-accumulate

Mnemonic

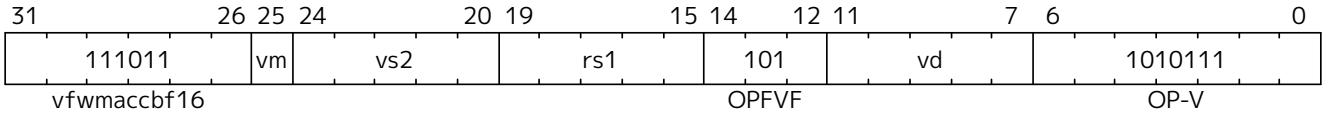
vfwmaccbf16.vv vd, vs1, vs2, vm

vfwmaccbf16.vf vd, rs1, vs2, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 16

Arguments

Register	Direction	EEW	Definition
Vd	input	32	FP32 Accumulate
Vs1/rs1	input	16	BF16 Source
Vs2	input	16	BF16 Source
Vd	output	32	FP32 Result

Description

This instruction performs a widening fused multiply-accumulate operation, where each pair of BF16 values are multiplied and their unrounded product is added to the corresponding FP32 accumulate value. The sum is rounded according to the *frm* register.

In the vector-vector version, the BF16 elements are read from `vs1` and `vs2` and FP32 accumulate value is read from `vd`. The FP32 result is written to the destination register `vd`.

The vector-scalar version is similar, but instead of reading elements from `vs1`, a scalar BF16 value is read from the FPU register `rs1`.

Exceptions: Overflow, Underflow, Inexact, Invalid

Operation

This `vfwmaccbf16.vv` instruction is equivalent to widening each of the BF16 inputs to FP32 and then performing an FMACC as shown in the following instruction sequence:

```
vfwcvtbf16.f.f.v T1, vs1, vm
vfwcvtbf16.f.f.v T2, vs2, vm
```

```
vfmacc.vv        vd, T1, T2, vm
```

Likewise, `vfwmaccbf16.vf` is equivalent to the following instruction sequence:

```
fcvt.s.bf16      T1, rs1
vfwcvtbf16.f.f.v T2, vs2, vm
vfmacc.vf        vd, T1, T2, vm
```

Included in: Zvfbfwma

Bibliography

754-2019 - IEEE Standard for Floating-Point Arithmetic
754-2008 - IEEE Standard for Floating-Point Arithmetic

Chapter 26. "Zfa" Extension for Additional Floating-Point Instructions, Version 1.0

This chapter describes the Zfa standard extension, which adds instructions for immediate loads, IEEE 754-2019 minimum and maximum operations, round-to-integer operations, and quiet floating-point comparisons. For RV32D, the Zfa extension also adds instructions to transfer double-precision floating-point values to and from integer registers, and for RV64Q, it adds analogous instructions for quad-precision floating-point values. The Zfa extension depends on the F extension.

26.1. Load-Immediate Instructions

The FLI.S instruction loads one of 32 single-precision floating-point constants, encoded in the *rs1* field, into floating-point register *rd*. The correspondence of *rs1* field values and single-precision floating-point values is shown in Table 35. FLI.S is encoded like FMV.W.X, but with *rs2*=1.

Table 35. Immediate values loaded by the FLI.S instruction.

rs1	Value	Sign	Exponent	Significand
0	−1.0	1	01111111	000…000
1	*Minimum positive normal*	0	00000001	000…000
2	1.0×2^{-16}	0	01101111	000…000
3	1.0×2^{-15}	0	01110000	000…000
4	1.0×2^{-8}	0	01110111	000…000
5	1.0×2^{-7}	0	01111000	000…000
6	0.0625 (2^{-4})	0	01111011	000…000
7	0.125 (2^{-3})	0	01111100	000…000
8	0.25	0	01111101	000…000
9	0.3125	0	01111101	010…000
10	0.375	0	01111101	100…000
11	0.4375	0	01111101	110…000
12	0.5	0	01111110	000…000
13	0.625	0	01111110	010…000
14	0.75	0	01111110	100…000
15	0.875	0	01111110	110…000
16	1.0	0	01111111	000…000
17	1.25	0	01111111	010…000
18	1.5	0	01111111	100…000
19	1.75	0	01111111	110…000
20	2.0	0	10000000	000…000
21	2.5	0	10000000	010…000
22	3	0	10000000	100…000
23	4	0	10000001	000…000
24	8	0	10000010	000…000

rs1	Value	Sign	Exponent	Significand
25	16	`0`	`10000011`	`000…000`
26	128 (2^7)	`0`	`10000110`	`000…000`
27	256 (2^8)	`0`	`10000111`	`000…000`
28	2^{15}	`0`	`10001110`	`000…000`
29	2^{16}	`0`	`10001111`	`000…000`
30	$+\infty$	`0`	`11111111`	`000…000`
31	*Canonical NaN*	`0`	`11111111`	`100…000`

The preferred assembly syntax for entries 1, 30, and 31 is `min`, `inf`, *and* `nan`, *respectively. For entries 0 through 29 (including entry 1), the assembler will accept decimal constants in C-like syntax.*

The set of 32 constants was chosen by examining floating-point libraries, including the C standard math library, and to optimize fixed-point to floating-point conversion.

Entries 8-22 follow a regular encoding pattern. No entry sets mantissa bits other than the two most significant ones.

If the D extension is implemented, FLI.D performs the analogous operation, but loads a double-precision value into floating-point register *rd*. Note that entry 1 (corresponding to the minimum positive normal value) has a numerically different value for double-precision than for single-precision. FLI.D is encoded like FLI.S, but with *fmt*=D.

If the Q extension is implemented, FLI.Q performs the analogous operation, but loads a quad-precision value into floating-point register *rd*. Note that entry 1 (corresponding to the minimum positive normal value) has a numerically different value for quad-precision. FLI.Q is encoded like FLI.S, but with *fmt*=Q.

If the Zfh or Zvfh extension is implemented, FLI.H performs the analogous operation, but loads a half-precision floating-point value into register *rd*. Note that entry 1 (corresponding to the minimum positive normal value) has a numerically different value for half-precision. Furthermore, since 2^{16} is not representable in half-precision floating-point, entry 29 in the table instead loads positive infinity—i.e., it is redundant with entry 30. FLI.H is encoded like FLI.S, but with *fmt*=H.

Additionally, since 2^{-16} and 2^{-15} are subnormal in half-precision, entry 1 is numerically greater than entries 2 and 3 for FLI.H.

The FLI.*fmt* instructions never set any floating-point exception flags.

26.2. Minimum and Maximum Instructions

The FMINM.S and FMAXM.S instructions are defined like the FMIN.S and FMAX.S instructions, except that if either input is NaN, the result is the canonical NaN.

If the D extension is implemented, FMINM.D and FMAXM.D instructions are analogously defined to operate on double-precision numbers.

If the Zfh extension is implemented, FMINM.H and FMAXM.H instructions are analogously defined to operate on half-precision numbers.

If the Q extension is implemented, FMINM.Q and FMAXM.Q instructions are analogously defined to operate on quad-precision numbers.

These instructions are encoded like their FMIN and FMAX counterparts, but with instruction bit 13 set to 1.

These instructions implement the IEEE 754-2019 minimum and maximum operations.

26.3. Round-to-Integer Instructions

The FROUND.S instruction rounds the single-precision floating-point number in floating-point register *rs1* to an integer, according to the rounding mode specified in the instruction's *rm* field. It then writes that integer, represented as a single-precision floating-point number, to floating-point register *rd*. Zero and infinite inputs are copied to *rd* unmodified. Signaling NaN inputs cause the invalid operation exception flag to be set; no other exception flags are set. FROUND.S is encoded like FCVT.S.D, but with *rs2*=4.

The FROUNDNX.S instruction is defined similarly, but it also sets the inexact exception flag if the input differs from the rounded result and is not NaN. FROUNDNX.S is encoded like FCVT.S.D, but with *rs2*=5.

If the D extension is implemented, FROUND.D and FROUNDNX.D instructions are analogously defined to operate on double-precision numbers. They are encoded like FCVT.D.S, but with *rs2*=4 and 5, respectively,

If the Zfh extension is implemented, FROUND.H and FROUNDNX.H instructions are analogously defined to operate on half-precision numbers. They are encoded like FCVT.H.S, but with *rs2*=4 and 5, respectively,

If the Q extension is implemented, FROUND.Q and FROUNDNX.Q instructions are analogously defined to operate on quad-precision numbers. They are encoded like FCVT.Q.S, but with *rs2*=4 and 5, respectively,

The FROUNDNX.fmt instructions implement the IEEE 754-2019 roundToIntegralExact operation, and the FROUND.fmt instructions implement the other operations in the roundToIntegral family.

26.4. Modular Convert-to-Integer Instruction

The FCVTMOD.W.D instruction is defined similarly to the FCVT.W.D instruction, with the following differences. FCVTMOD.W.D always rounds towards zero. Bits 31:0 are taken from the rounded, unbounded two's complement result, then sign-extended to XLEN bits and written to integer register *rd*. $\pm\infty$ and NaN are converted to zero.

Floating-point exception flags are raised the same as they would be for FCVT.W.D with the same input operand.

This instruction is only provided if the D extension is implemented. It is encoded like FCVT.W.D, but with the rs2 field set to 8 and the *rm* field set to 1 (RTZ). Other *rm* values are *reserved*.

The assembly syntax requires the RTZ rounding mode to be explicitly specified, i.e., `fcvtmod.w.d rd, rs1, rtz`.

The FCVTMOD.W.D instruction was added principally to accelerate the processing of JavaScript Numbers. Numbers are double-precision values, but some operators implicitly truncate them to signed integers mod 2^{32}.

26.5. Move Instructions

For RV32 only, if the D extension is implemented, the FMVH.X.D instruction moves bits 63:32 of floating-point register *rs1* into integer register *rd*. It is encoded in the OP-FP major opcode with *funct3*=0, *rs2*=1, and *funct7*=1110001.

FMVH.X.D is used in conjunction with the existing FMV.X.W instruction to move a double-precision floating-point number to a pair of x-registers.

For RV32 only, if the D extension is implemented, the FMVP.D.X instruction moves a double-precision number from a pair of integer registers into a floating-point register. Integer registers *rs1* and *rs2* supply bits 31:0 and 63:32, respectively; the result is written to floating-point register *rd*. FMVP.D.X is encoded in the OP-FP major opcode with *funct3*=0 and *funct7*=1011001.

For RV64 only, if the Q extension is implemented, the FMVH.X.Q instruction moves bits 127:64 of floating-point register *rs1* into integer register *rd*. It is encoded in the OP-FP major opcode with *funct3*=0, *rs2*=1, and *funct7*=1110011.

FMVH.X.Q is used in conjunction with the existing FMV.X.D instruction to move a quad-precision floating-point number to a pair of x-registers.

For RV64 only, if the Q extension is implemented, the FMVP.Q.X instruction moves a double-precision number from a pair of integer registers into a floating-point register. Integer registers *rs1* and *rs2* supply bits 63:0 and 127:64, respectively; the result is written to floating-point register *rd*. FMVP.Q.X is encoded in the OP-FP major opcode with *funct3*=0 and *funct7*=1011011.

26.6. Comparison Instructions

The FLEQ.S and FLTQ.S instructions are defined like the FLE.S and FLT.S instructions, except that quiet NaN inputs do not cause the invalid operation exception flag to be set.

If the D extension is implemented, FLEQ.D and FLTQ.D instructions are analogously defined to operate on double-precision numbers.

If the Zfh extension is implemented, FLEQ.H and FLTQ.H instructions are analogously defined to operate on half-precision numbers.

If the Q extension is implemented, FLEQ.Q and FLTQ.Q instructions are analogously defined to operate on quad-precision numbers.

These instructions are encoded like their FLE and FLT counterparts, but with instruction bit 14 set to 1.

We do not expect analogous comparison instructions will be added to the vector ISA, since they can be reasonably efficiently emulated using masking.

Chapter 27. "Zfinx", "Zdinx", "Zhinx", "Zhinxmin" Extensions for Floating-Point in Integer Registers, Version 1.0

This chapter defines the "Zfinx" extension (pronounced "z-f-in-x") that provides instructions similar to those in the standard floating-point F extension for single-precision floating-point instructions but which operate on the `x` registers instead of the `f` registers. This chapter also defines the "Zdinx", "Zhinx", and "Zhinxmin" extensions that provide similar instructions for other floating-point precisions.

The F extension uses separate `f` registers for floating-point computation, to reduce register pressure and simplify the provision of register-file ports for wide superscalars. However, the additional architectural state increases the minimal implementation cost. By eliminating the `f` registers, the Zfinx extension substantially reduces the cost of simple RISC-V implementations with floating-point instruction-set support. Zfinx also reduces context-switch cost.

In general, software that assumes the presence of the F extension is incompatible with software that assumes the presence of the Zfinx extension, and vice versa.

The Zfinx extension adds all of the instructions that the F extension adds, *except* for the transfer instructions FLW, FSW, FMV.W.X, FMV.X.W, C.FLW[SP], and C.FSW[SP].

Zfinx software uses integer loads and stores to transfer floating-point values from and to memory. Transfers between registers use either integer arithmetic or floating-point sign-injection instructions.

The Zfinx variants of these F-extension instructions have the same semantics, except that whenever such an instruction would have accessed an `f` register, it instead accesses the `x` register with the same number.

The Zfinx extension depends on the "Zicsr" extension for control and status register access.

27.1. Processing of Narrower Values

Floating-point operands of width w < XLEN bits occupy bits w-1:0 of an `x` register. Floating-point operations on w-bit operands ignore operand bits XLEN-1: w.

Floating-point operations that produce w < XLEN-bit results fill bits XLEN-1: w with copies of bit w-1 (the sign bit).

The NaN-boxing scheme employed in the `f` registers was designed to efficiently support recoded floating-point formats. Recoding is less practical for Zfinx, though, since the same registers hold both floating-point and integer operands. Hence, the need for NaN boxing is diminished.

Sign-extending 32-bit floating-point numbers when held in RV64 `x` registers is compatible with the existing RV64 calling conventions, which leave bits 63-32 undefined when passing a 32-bit floating point value in `x` registers. To keep the architecture more regular, we extend this pattern to 16-bit floating-point numbers in both RV32 and RV64.

27.2. Zdinx

The Zdinx extension provides analogous double-precision floating-point instructions. The Zdinx extension depends upon the Zfinx extension.

The Zdinx extension adds all of the instructions that the D extension adds, *except* for the transfer

instructions FLD, FSD, FMV.D.X, FMV.X.D, C.FLD[SP], and C.FSD[SP].

The Zdinx variants of these D-extension instructions have the same semantics, except that whenever such an instruction would have accessed an `f` register, it instead accesses the `x` register with the same number.

27.3. Processing of Wider Values

Double-precision operands in RV32Zdinx are held in aligned `x`-register pairs, i.e., register numbers must be even. Use of misaligned (odd-numbered) registers for double-width floating-point 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 double-precision operand in RV32Zdinx might be held in register `x14`, with bits 63:32 of that operand held in `x15`.

When a double-width floating-point result is written to `x0`, the entire write takes no effect: e.g., for RV32Zdinx, writing a double-precision result to `x0` does not cause `x1` to be written.

When `x0` is used as a double-width floating-point operand, the entire operand is zero—i.e., `x1` is not accessed.

Load-pair and store-pair instructions are contained in a separate extension (see Section Extensions for Load/Store pair for RV32). In case this is not available, transferring double-precision operands in RV32Zdinx from or to memory requires two loads or stores. Register moves need only a single FSGNJ.D instruction, however.

27.4. Zhinx

The Zhinx extension provides analogous half-precision floating-point instructions. The Zhinx extension depends upon the Zfinx extension.

The Zhinx extension adds all of the instructions that the Zfh extension adds, *except* for the transfer instructions FLH, FSH, FMV.H.X, and FMV.X.H.

The Zhinx variants of these Zfh-extension instructions have the same semantics, except that whenever such an instruction would have accessed an `f` register, it instead accesses the `x` register with the same number.

27.5. Zhinxmin

The Zhinxmin extension provides minimal support for 16-bit half-precision floating-point instructions that operate on the `x` registers. The Zhinxmin extension depends upon the Zfinx extension.

The Zhinxmin extension includes the following instructions from the Zhinx extension: FCVT.S.H and FCVT.H.S. If the Zdinx extension is present, the FCVT.D.H and FCVT.H.D instructions are also included.

In the future, an RV64Zqinx quad-precision extension could be defined analogously to RV32Zdinx. An RV32Zqinx extension could also be defined but would require quad-register groups.

27.6. Privileged Architecture Implications

In the standard privileged architecture defined in Volume II, the `mstatus` field FS is hardwired to 0 if the

Zfinx extension is implemented, and FS no longer affects the trapping behavior of floating-point instructions or `fcsr` accesses.

The `misa` bits F, D, and Q are hardwired to 0 when the Zfinx extension is implemented.

A future discoverability mechanism might be used to probe the existence of the Zfinx, Zhinx, and Zdinx extensions.

Chapter 28. "C" Extension for Compressed Instructions, Version 2.0

This chapter describes the RISC-V standard compressed instruction-set extension, named "C", which reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32I, RV32E, RV64I, RV64E), and we use the generic term "RVC" to cover any of these. Typically, 50%-60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%-30% code-size reduction.

28.1. Overview

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions.

Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.

The compressed instruction encodings are mostly common across RV32C and RV64C, but as shown in Figure 3, a few opcodes are used for different purposes depending on base ISA. For example, the wider address-space RV64C variant requires additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C.

Double-precision loads and stores are a significant fraction of static and dynamic instructions, hence the motivation to include them in the RV32C and RV64C encoding.

Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.

Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.

Although reusing opcodes for different purposes for different base ISAs adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISAs. The compressed floating-point load and store variants use the

same instruction format with the same register specifiers as the wider integer loads and stores.

RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E or RV64I/E) or the F and D standard extensions where present. Adopting this constraint has two main benefits:

- Hardware designs can simply expand RVC instructions during decode, simplifying verification and minimizing modifications to existing microarchitectures.
- Compilers can be unaware of the RVC extension and leave code compression to the assembler and linker, although a compression-aware compiler will generally be able to produce better results.

We felt the multiple complexity reductions of a simple one-one mapping between C and base IFD instructions far outweighed the potential gains of a slightly denser encoding that added additional instructions only supported in the C extension, or that allowed encoding of multiple IFD instructions in one C instruction.

It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA.

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch (Buchholz, 1962), developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture (Amdahl et al., 1964) supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 (Thornton, 1965), a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25-30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.

Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to

significant complexity in documentation, implementations, and software tools support.

Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size. (Waterman, 2011)

28.2. Compressed Instruction Formats

Table 36 shows the nine compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, CA, and CB are limited to just 8 of them. Table 37 lists these popular registers, which correspond to registers `x8` to `x15`. Note that there is a separate version of load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow access to all 32 data registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.

The RISC-V ABI was changed to make the frequently used registers map to registers 'x8-x15'. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E and RV64E base ISAs, which only have 16 integer registers.

Compressed register-based floating-point loads and stores also use the CL and CS formats respectively, with the eight registers mapping to `f8` to `f15`.

The standard RISC-V calling convention maps the most frequently used floating-point registers to registers `f8` to `f15`, which allows the same register decompression decoding as for integer register numbers.

The formats were designed to keep bits for the two register source specifiers in the same place in all instructions, while the destination register field can move. When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding. Where immediates are sign-extended, the sign extension is always from bit 12. Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate multiplexers required.

The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations.

For many RVC instructions, zero-valued immediates are disallowed and `x0` is not a valid 5-bit register specifier. These restrictions free up encoding space for other instructions requiring fewer operand bits.

Table 36. Compressed 16-bit RVC instruction formats

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CA	Arithmetic	funct6			rd′/rs1′	funct2	rs2′	op
CB	Branch/Arithmetic	funct3	offset		rd′/rs1′	offset		op
CJ	Jump	funct3	jump target					op

Table 37. Registers specified by the three-bit rs1′, rs2′, and rd′ fields of the CIW, CL, CS, CA, and CB formats.

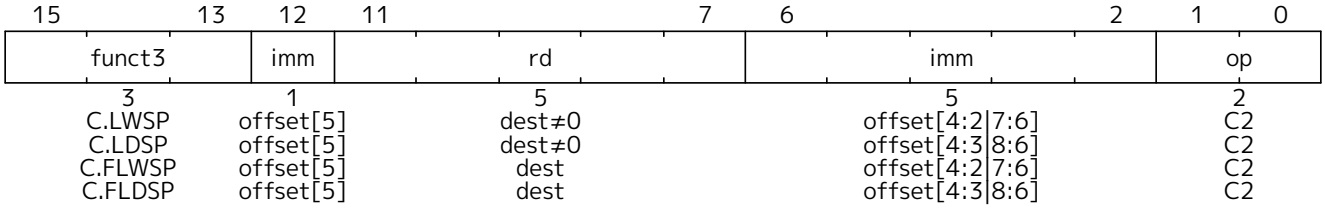
RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

28.3. Load and Store Instructions

To increase the reach of 16-bit instructions, data-transfer instructions use zero-extended immediates that are scaled by the size of the data in bytes: ×4 for words, ×8 for double words, and ×16 for quad words.

RVC provides two variants of loads and stores. One uses the ABI stack pointer, `x2`, as the base address and can target any data register. The other can reference one of 8 base address registers and one of 8 data registers.

28.3.1. Stack-Pointer-Based Loads and Stores



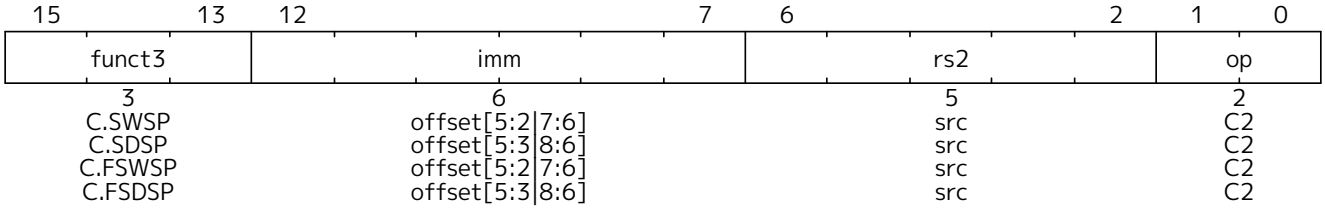
These instructions use the CI format.

C.LWSP loads a 32-bit value from memory into register *rd*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `lw rd, offset(x2)`. C.LWSP is valid only when *rd*≠`x0`; the code points with *rd*=`x0` are reserved.

C.LDSP is an RV64C-only instruction that loads a 64-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `ld rd, offset(x2)`. C.LDSP is valid only when *rd*≠`x0`; the code points with *rd*=`x0` are reserved.

C.FLWSP is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `flw rd, offset(x2)`.

C.FLDSP is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `fld rd, offset(x2)`.



These instructions use the CSS format.

C.SWSP stores a 32-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `sw rs2, offset(x2)`.

C.SDSP is an RV64C-only instruction that stores a 64-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `sd rs2, offset(x2)`.

C.FSWSP is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `fsw rs2, offset(x2)`.

C.FSDSP is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `fsd rs2, offset(x2)`.

Register save/restore code at function entry/exit represents a significant portion of static code size. The stack-pointer-based compressed loads and stores in RVC are effective at reducing the save/restore static code size by a factor of 2 while improving performance by reducing dynamic instruction bandwidth.

A common mechanism used in other ISAs to further reduce save/restore code size is load-multiple and store-multiple instructions. We considered adopting these for RISC-V but noted the following drawbacks to these instructions:

- *These instructions complicate processor implementations.*
- *For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions.*
- *Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple.*
- *Unlike the rest of the RVC instructions, the compiler would have to be aware of these load-multiple and store-multiple instructions to both allocate registers in the expected order and also to schedule the loads and stores contiguously and in the proper order, to maximize the chances of them being detected and replaced by an assembler or linker with the equivalent load-multiple or store-multiple compressed instruction.*
- *Simple microarchitectural implementations will constrain how other instructions can be scheduled around the load and store multiple instructions, leading to a potential*

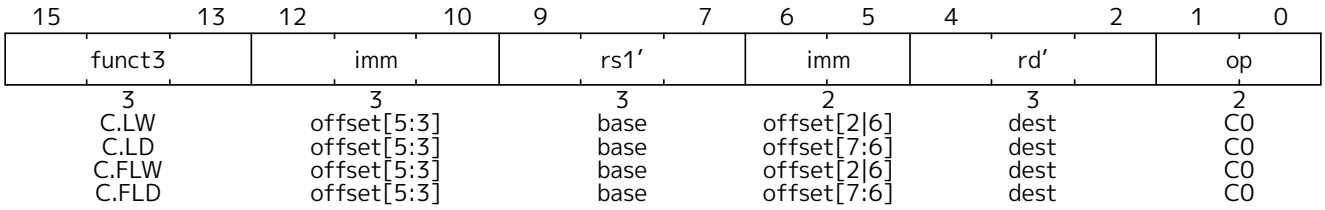
performance loss.

- *The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, CA, and CB formats.*

Furthermore, much of the gains can be realized in software by replacing prologue and epilogue code with subroutine calls to common prologue and epilogue code, a technique described in Section 5.6 of (Waterman, 2016).

While reasonable architects might come to different conclusions, we decided to omit load and store multiple and instead use the software-only approach of calling save/restore millicode routines to attain the greatest code size reduction.

28.3.2. Register-Based Loads and Stores



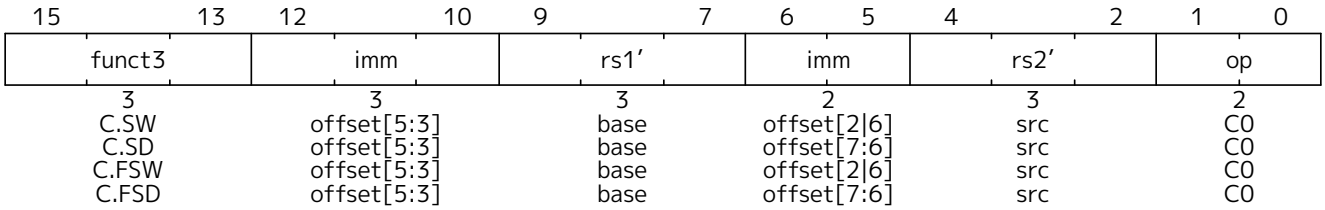
These instructions use the CL format.

C.LW loads a 32-bit value from memory into register ***rd′***. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register ***rs1′***. It expands to `lw rd′, offset(rs1′)`.

C.LD is an RV64C-only instruction that loads a 64-bit value from memory into register ***rd′***. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register ***rs1′***. It expands to `ld rd′, offset(rs1′)`.

C.FLW is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register ***rd′***. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register ***rs1′***. It expands to `flw rd′, offset(rs1′)`.

C.FLD is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register ***rd′***. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register ***rs1′***. It expands to `fld rd′, offset(rs1′)`.



These instructions use the CS format.

C.SW stores a 32-bit value in register ***rs2′*** to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register ***rs1′***. It expands to `sw rs2′, offset(rs1′)`.

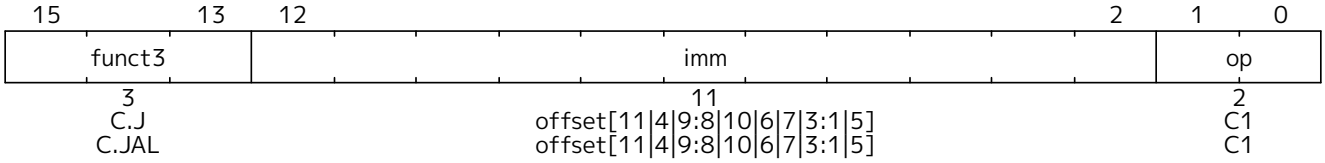
C.SD is an RV64C-only instruction that stores a 64-bit value in register ***rs2′*** to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register ***rs1′***. It expands to `sd rs2′, offset(rs1′)`.

C.FSW is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `fsw rs2′, offset(rs1′)`.

C.FSD is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `fsd rs2′, offset(rs1′)`.

28.4. Control Transfer Instructions

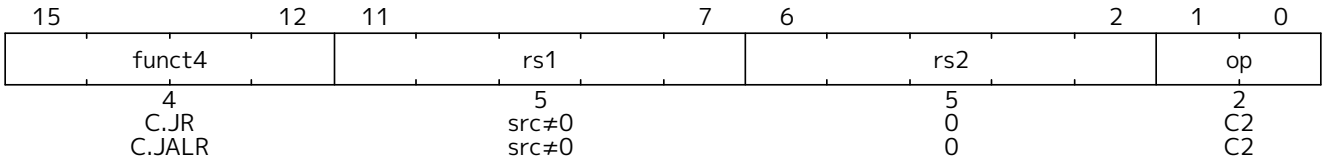
RVC provides unconditional jump instructions and conditional branch instructions. As with base RVI instructions, the offsets of all RVC control transfer instructions are in multiples of 2 bytes.



These instructions use the CJ format.

C.J performs an unconditional control transfer. The offset is sign-extended and added to the `pc` to form the jump target address. C.J can therefore target a ±2 KiB range. C.J expands to `jal x0, offset`.

C.JAL is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (`pc`+2) to the link register, `x1`. C.JAL expands to `jal x1, offset`.

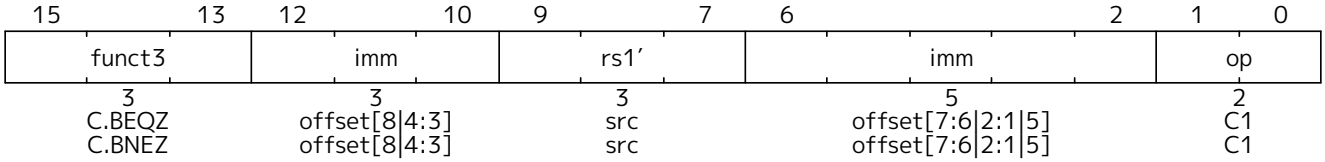


These instructions use the CR format.

C.JR (jump register) performs an unconditional control transfer to the address in register *rs1*. C.JR expands to `jalr x0, 0(rs1)`. C.JR is valid only when *rs1*≠`x0`; the code point with *rs1*=`x0` is reserved.

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (`pc`+2) to the link register, `x1`. C.JALR expands to `jalr x1, 0(rs1)`. C.JALR is valid only when *rs1*≠`x0`; the code point with *rs1*=`x0` corresponds to the C.EBREAK instruction.

Strictly speaking, C.JALR does not expand exactly to a base RVI instruction as the value added to the PC to form the link address is 2 rather than 4 as in the base ISA, but supporting both offsets of 2 and 4 bytes is only a very minor change to the base microarchitecture.



These instructions use the CB format.

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the `pc` to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register *rs1′* is zero. It expands to `beq rs1′, x0, offset`.

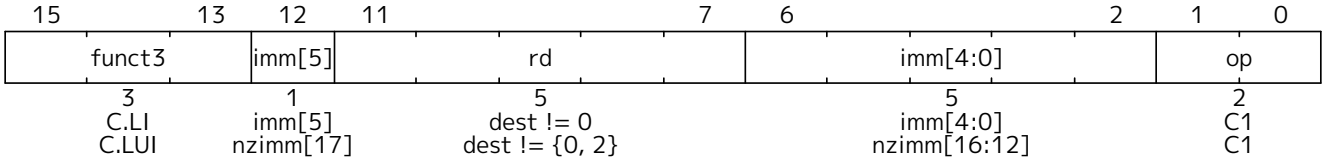
C.BNEZ is defined analogously, but it takes the branch if *rs1′* contains a nonzero value. It expands to `bne rs1′, x0, offset`.

28.5. Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

28.5.1. Integer Constant-Generation Instructions

The two constant-generation instructions both use the CI instruction format and can target any integer register.

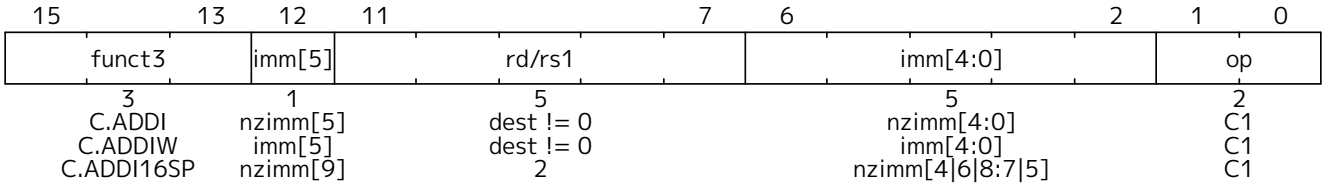


C.LI loads the sign-extended 6-bit immediate, *imm*, into register *rd*. C.LI expands into `addi rd, x0, imm`. The C.LI code points with *rd*=`x0` are HINTs.

C.LUI loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI expands into `lui rd, imm`. C.LUI is valid only when *rd*≠`x2`, and when the immediate is not equal to zero. The code points with *imm*=0 are reserved. The code points with *rd*=`x2` and *imm*≠0 correspond to the C.ADDI16SP instruction. The code points with *rd*=`x0` and *imm*≠0 are HINTs.

28.5.2. Integer Register-Immediate Operations

These integer register-immediate operations are encoded in the CI format and perform operations on an integer register and a 6-bit immediate.

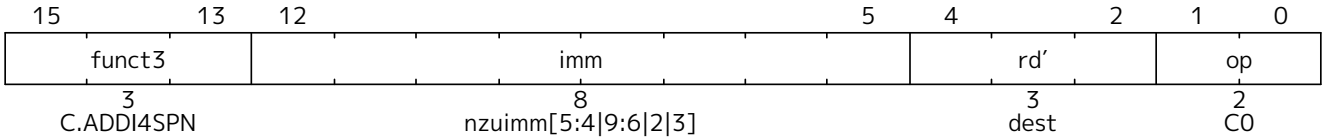


C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register *rd* then writes the result to *rd*. C.ADDI expands into `addi rd, rd, imm`. The code points with *rd*≠0 and *imm*=0 are HINTs. The code points with *rd*=`x0` encode the C.NOP instruction, of which the code points with *imm*≠0 are HINTs.

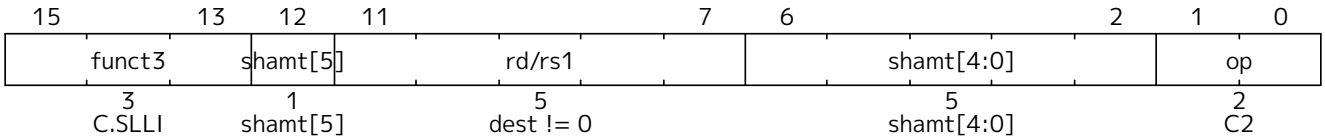
C.ADDIW is an RV64C-only instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into `addiw rd, rd, imm`. The immediate can be zero for C.ADDIW, where this corresponds to `sext.w rd`. C.ADDIW is valid only when *rd*≠`x0`; the code points with *rd*=`x0` are reserved.

C.ADDI16SP (add immediate to stack pointer) shares the opcode with C.LUI, but has a destination field of `x2`. C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (`sp=x2`), where the immediate is scaled to represent multiples of 16 in the range [-512, 496]. C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into `addi x2, x2, nzimm[9:4]`. C.ADDI16SP is valid only when *nzimm*≠0; the code point with *nzimm*=0 is reserved.

In the standard RISC-V calling convention, the stack pointer `sp` *is always 16-byte aligned.*



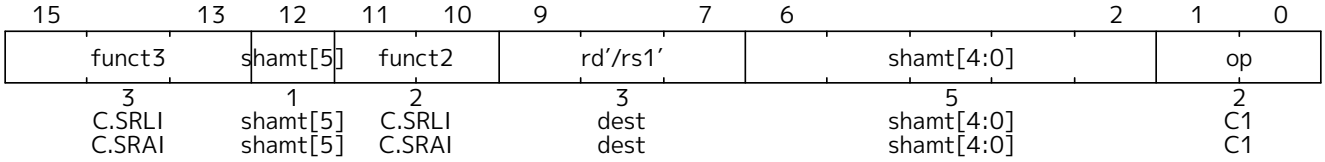
C.ADDI4SPN (add immediate to stack pointer, non-destructive) is a CIW-format instruction that adds a *zero*-extended non-zero immediate, scaled by 4, to the stack pointer, `x2`, and writes the result to `rd'`. This instruction is used to generate pointers to stack-allocated variables, and expands to `addi rd', x2, nzuimm[9:2]`. C.ADDI4SPN is valid only when *nzuimm*≠0; the code points with *nzuimm*=0 are reserved.



C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field. C.SLLI expands into `slli rd, rd, shamt[5:0]`.

The C.SLLI code points with *shamt*=0 or with *rd*=`x0` are HINTs.

For RV32C, *shamt[5]* must be zero; the code points with *shamt[5]*=1 are designated for custom extensions.



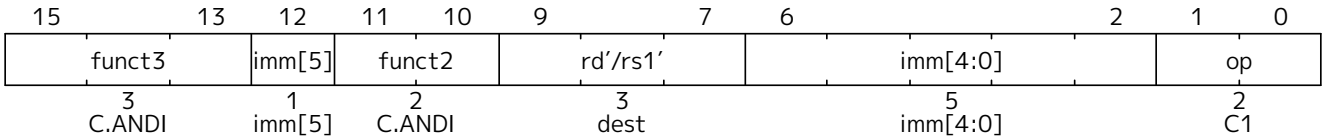
C.SRLI is a CB-format instruction that performs a logical right shift of the value in register *rd'* then writes the result to *rd'*. The shift amount is encoded in the *shamt* field. C.SRLI expands into `srli rd', rd', shamt`.

The C.SRLI code points with *shamt*=0 are HINTs.

For RV32C, *shamt[5]* must be zero; the code points with *shamt[5]*=1 are designated for custom extensions.

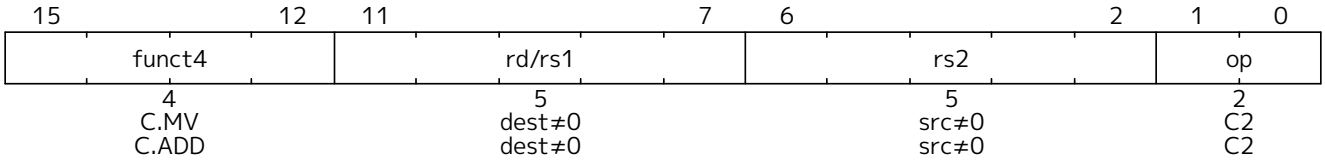
C.SRAI is defined analogously to C.SRLI, but instead performs an arithmetic right shift. C.SRAI expands to `srai rd', rd', shamt`.

Left shifts are usually more frequent than right shifts, as left shifts are frequently used to scale address values. Right shifts have therefore been granted less encoding space and are placed in an encoding quadrant where all other immediates are sign-extended.



C.ANDI is a CB-format instruction that computes the bitwise AND of the value in register *rd'* and the sign-extended 6-bit immediate, then writes the result to *rd'*. C.ANDI expands to `andi rd', rd', imm`.

28.5.3. Integer Register-Register Operations

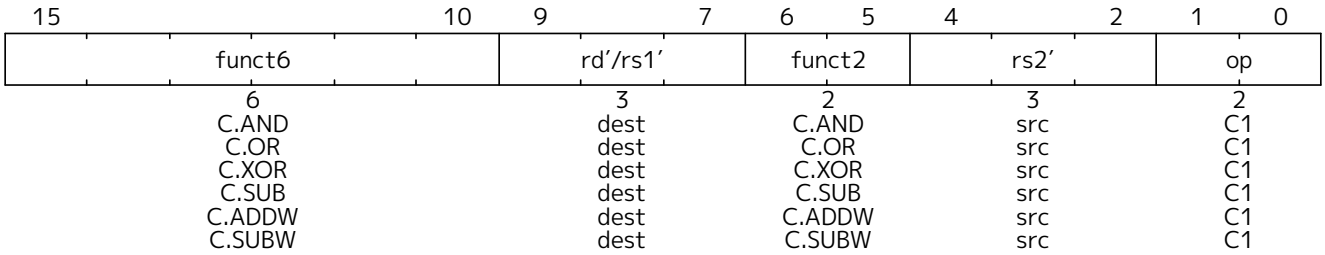


These instructions use the CR format.

C.MV copies the value in register *rs2* into register *rd*. C.MV expands into `add rd, x0, rs2`. C.MV is valid only when *rs2*≠`x0`; the code points with *rs2*=`x0` correspond to the C.JR instruction. The code points with *rs2*≠`x0` and *rd*=`x0` are HINTs.

C.MV expands to a different instruction than the canonical MV pseudoinstruction, which instead uses ADDI. Implementations that handle MV specially, e.g. using register-renaming hardware, may find it more convenient to expand C.MV to MV instead of ADD, at slight additional hardware cost.

C.ADD adds the values in registers *rd* and *rs2* and writes the result to register *rd*. C.ADD expands into `add rd, rd, rs2`. C.ADD is only valid when *rs2*≠`x0`; the code points with *rs2*=`x0` correspond to the C.JALR and C.EBREAK instructions. The code points with *rs2*≠`x0` and *rd*=`x0` are HINTs.



These instructions use the CA format.

`C.AND` computes the bitwise `AND` of the values in registers *rd'* and *rs2'*, then writes the result to register *rd'*. `C.AND` expands into `and rd', rd', rs2'`.

`C.OR` computes the bitwise `OR` of the values in registers *rd'* and *rs2'*, then writes the result to register *rd'*. `C.OR` expands into `or rd', rd', rs2'`.

`C.XOR` computes the bitwise `XOR` of the values in registers *rd'* and *rs2'*, then writes the result to register *rd'*.

`C.XOR` expands into `xor rd', rd', rs2'`.

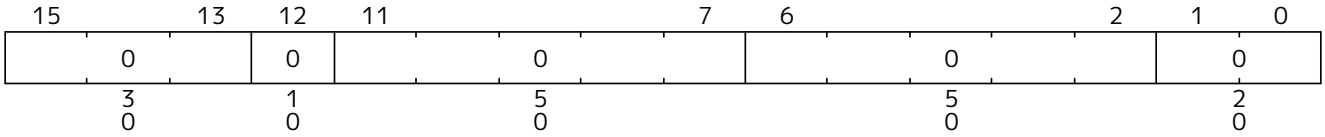
`C.SUB` subtracts the value in register *rs2'* from the value in register *rd'*, then writes the result to register *rd'*. `C.SUB` expands into `sub rd', rd', rs2'`.

`C.ADDW` is an RV64C-only instruction that adds the values in registers *rd'* and *rs2'*, then sign-extends the lower 32 bits of the sum before writing the result to register *rd'*. `C.ADDW` expands into `addw rd', rd', rs2'`.

`C.SUBW` is an RV64C-only instruction that subtracts the value in register *rs2'* from the value in register *rd'*, then sign-extends the lower 32 bits of the difference before writing the result to register *rd'*. `C.SUBW` expands into `subw rd', rd', rs2'`.

This group of six instructions do not provide large savings individually, but do not occupy much encoding space and are straightforward to implement, and as a group provide a worthwhile improvement in static and dynamic compression.

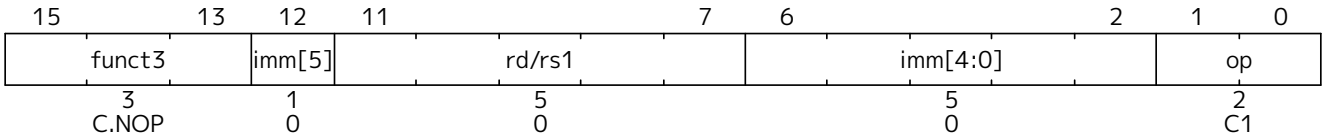
28.5.4. Defined Illegal Instruction



A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.

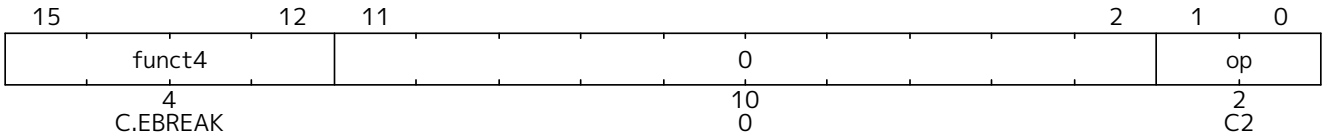
We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zeroed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very long instructions in the RISC-V variable-length encoding scheme) as illegal to capture another common value seen in non-existent memory regions.

28.5.5. NOP Instruction



`C.NOP` is a CI-format instruction that does not change any user-visible state, except for advancing the `pc` and incrementing any applicable performance counters. `C.NOP` expands to `nop`. The `C.NOP` code points with *imm*≠0 encode HINTs.

28.5.6. Breakpoint Instruction



Debuggers can use the `C.EBREAK` instruction, which expands to `ebreak`, to cause control to be transferred back to the debugging environment. `C.EBREAK` shares the opcode with the `C.ADD` instruction, but with *rd* and *rs2* both zero, thus can also use the `CR` format.

28.6. Usage of C Instructions in LR/SC Sequences

On implementations that support the C extension, compressed forms of the I instructions permitted inside constrained LR/SC sequences, as described in Section 13.3, are also permitted inside constrained LR/SC sequences.

The implication is that any implementation that claims to support both the A and C extensions must ensure that LR/SC sequences containing valid C instructions will eventually complete.

28.7. HINT Instructions

A portion of the RVC encoding space is reserved for microarchitectural HINTs. Like the HINTs in the RV32I base ISA (see HINT Instructions), these instructions do not modify any architectural state, except for advancing the `pc` and any applicable performance counters. HINTs are executed as no-ops on implementations that ignore them.

RVC HINTs are encoded as computational instructions that do not modify the architectural state, either because *rd*=`x0` (e.g. `C.ADD x0, t0`), or because *rd* is overwritten with a copy of itself (e.g. `C.ADDI t0, 0`).

This HINT encoding has been chosen so that simple implementations can ignore HINTs altogether, and instead execute a HINT as a regular computational instruction that happens not to mutate the architectural state.

RVC HINTs do not necessarily expand to their RVI HINT counterparts. For example, `C.ADD` *x0*, *a0* might not encode the same HINT as `ADD` *x0*, *x0*, *a0*.

The primary reason to not require an RVC HINT to expand to an RVI HINT is that HINTs are unlikely to be compressible in the same manner as the underlying computational instruction. Also, decoupling the RVC and RVI HINT mappings allows the scarce RVC HINT space to be allocated to the most popular HINTs, and in particular, to HINTs that are amenable to macro-op fusion.

Table 38 lists all RVC HINT code points. For RV32C, 78% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs; no standard HINTs will ever be defined in this subspace.

Table 38. RVC HINT instructions.

Instruction	Constraints	Code Points	Purpose
C.NOP	*imm*≠0	63	*Designated for future standard use*
C.ADDI	*rd*≠`x0`, *imm*=0	31	
C.LI	*rd*=`x0`	64	
C.LUI	*rd*=`x0`, *imm*≠0	63	
C.MV	*rd*=`x0`, *rs2*≠`x0`	31	
C.ADD	*rd*=`x0`, *rs2*≠`x0`, *rs2*≠`x2-x5`	27	

Instruction	Constraints	Code Points	Purpose
C.ADD	*rd*=`x0`, *rs2*=`x2-x5`	4	(rs2=x2) C.NTL.P1 (rs2=x3) C.NTL.PALL (rs2=x4) C.NTL.S1 (rs2=x5) C.NTL.ALL
C.SLLI	*rd*=`x0` or *imm*=0	63 (RV32), 95 (RV64)	*Designated for custom use*
C.SRLI	*imm*=0	8	
C.SRAI	*imm*=0	8	

28.8. RVC Instruction Set Listings

Table 39 shows a map of the major opcodes for RVC. Each row of the table corresponds to one quadrant of the encoding space. The last quadrant, which has the two least-significant bits set, corresponds to instructions wider than 16 bits, including those in the base ISAs. Several instructions are only valid for certain operands; when invalid, they are marked either *RES* to indicate that the opcode is reserved for future standard extensions; *Custom* to indicate that the opcode is designated for custom extensions; or *HINT* to indicate that the opcode is reserved for microarchitectural hints (see Section 28.7).

Table 39. RVC opcode map instructions.

inst[15:13] inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD	LW	FLW LD	*Reserved*	FSD FSD	SW	FSW SD	RV32 RV64
01	ADDI	JAL ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64
10	SLLI	FLDSP FLDSP	LWSP	FLWSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP	SWSP	FSWSP SDSP	RV32 RV64
11	>16b								

Figure 3, Figure 4, and Figure 5 list the RVC instructions.

15–13	12–10	9–7	6–5	4–2	1–0	
000	0			0	00	Illegal instruction
000	uimm[5:4|9:6|2|3]			rd′	00	*C.ADDI4SPN* *(RES, uimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	*C.FLW* *(RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	*C.LD* *(RV64)*
100	---				00	Reserved
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	*C.FSW* *(RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	*C.SD* *(RV64)*

Figure 3. Instruction listing for RVC, Quadrant 0

15–13	12	11–10	9–8	7–6	5–4	3–2	1–0	
000	imm[5]	0		imm[4:0]			01	*C.NOP (HINT, imm¬0)*
000	imm[5]	rs1/rd≠0		imm[4:0]			01	*C.ADDI (HINT, imm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]						01	*C.JAL (RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]			01	*C.ADDIW (RV64; RES, rd=0)*
010	imm[5]	rd≠0		imm[4:0]			01	*C.LI (HINT, rd=0)*
011	imm[9]	2		imm[4|6|8:7|5]			01	*C.ADDI16SP (RES, imm=0)*
011	imm[17]	rd≠{0, 2}		imm[16:12]			01	*C.LUI (RES, imm=0; HINT, rd=0)*
100	uimm[5]	00	rs1′/rd′	uimm[4:0]			01	*C.SRLI (HINT, uimm=0)*
100	uimm[5]	01	rs1′/rd′	uimm[4:0]			01	*C.SRAI (HINT, uimm=0)*
100	imm[5]	10	rs1′/rd′	imm[4:0]			01	C.ANDI
100	0	11	rs1′/rd′	00	rs2′		01	C.SUB
100	0	11	rs1′/rd′	01	rs2′		01	C.XOR
100	0	11	rs1′/rd′	10	rs2′		01	C.OR
100	0	11	rs1′/rd′	11	rs2′		01	C.AND
100	1	11	rs1′/rd′	00	rs2′		01	*C.SUBW (RV64; RV32 RES)*
100	1	11	rs1′/rd′	01	rs2′		01	*C.ADDW (RV64; RV32 RES)*
100	1	11	---	10	---		01	Reserved
100	1	11	---	11	---		01	Reserved
101	imm[11|4|9:8|10|6|7|3:1|5]						01	C.J
110	imm[8|4:3]		rs1′	imm[7:6|2:1|5]			01	C.BEQZ
111	imm[8|4:3]		rs1′	imm[7:6|2:1|5]			01	C.BNEZ

Figure 4. Instruction listing for RVC, Quadrant 1

15–13	12	11–10	9–8	7–6	5–4	3–2	1–0	
000	nzuimm[5]	rs1/rd≠0		nzuimm[4:0]			10	*C.SLLI (HINT, rd=0 or imm=0)*
001	uimm[5]	rd		uimm[4:3|8:6]			10	C.FLDSP
010	uimm[5]	rd≠0		uimm[4:2|7:6]			10	*C.LWSP (RES, rd=0)*
011	uimm[5]	rd		uimm[4:2|7:6]			10	*C.FLWSP (RV32)*
011	uimm[5]	rd≠0		uimm[4:3|8:6]			10	*C.LDSP (RV64; RES, rd=0)*
100	0	rs1≠0		0			10	*C.JR (RES, rs1=0)*
100	0	rd≠0		rs2≠0			10	*C.MV (HINT, rd=0)*
100	1	0		0			10	C.EBREAK
100	1	rs1≠0		0			10	C.JALR
100	1	rs1/rd≠0		rs2≠0			10	*C.ADD (HINT, rd=0)*
101	uimm[5:3|8:6]			rs2			10	C.FSDSP
110	uimm[5:2|7:6]			rs2			10	C.SWSP
111	uimm[5:2|7:6]			rs2			10	*C.FSWSP (RV32)*
111	uimm[5:3|8:6]			rs2			10	*C.SDSP (RV64)*

Figure 5. Instruction listing for RVC, Quadrant 2

Chapter 29. "Zc*" Extension for Code Size Reduction, Version 1.0.0

29.1. Zc* Overview

Zc* is a group of extensions that define subsets of the existing C extension (Zca, Zcd, Zcf) and new extensions which only contain 16-bit encodings.

Zcm* all reuse the encodings for *c.fld*, *c.fsd*, *c.fldsp*, *c.fsdsp*.

Table 40. Zc extension overview*

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores						
C excl. c.f*	yes					
The Zcf extension is added as a way to refer to compressed single-precision floating-point load/stores						
c.flw		rv32				
c.flwsp		rv32				
c.fsw		rv32				
c.fswsp		rv32				
The Zcd extension is added as a way to refer to compressed double-precision floating-point load/stores						
c.fld			yes			
c.fldsp			yes			
c.fsd			yes			
c.fsdsp			yes			
Simple operations for use on all architectures						
c.lbu				yes		
c.lh				yes		
c.lhu				yes		
c.sb				yes		
c.sh				yes		
c.zext.b				yes		
c.sext.b				yes		
c.zext.h				yes		
c.sext.h				yes		
c.zext.w				yes		
c.mul				yes		
c.not				yes		
PUSH/POP and double move which overlap with *c.fsdsp*. Complex operations intended for embedded CPUs						
cm.push					yes	
cm.pop					yes	
cm.popret					yes	

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
cm.popretz					yes	
cm.mva01s					yes	
cm.mvsa01					yes	
Table jump which overlaps with *c.fsdsp*. Complex operations intended for embedded CPUs						
cm.jt						yes
cm.jalt						yes

29.2. C

The C extension is the superset of the following extensions:

- Zca
- Zcf if F is specified (RV32 only)
- Zcd if D is specified

As C defines the same instructions as Zca, Zcf, and Zcd, the rule is that:

- C always implies Zca
- C+F implies Zcf (RV32 only)
- C+D implies Zcd

29.3. Zce

The Zce extension is intended to be used for microcontrollers, and includes all relevant Zc extensions.

- Specifying Zce on RV32 without F includes Zca, Zcb, Zcmp, Zcmt
- Specifying Zce on RV32 with F includes Zca, Zcb, Zcmp, Zcmt *and* Zcf
- Specifying Zce on RV64 always includes Zca, Zcb, Zcmp, Zcmt
  - Zcf doesn't exist for RV64

Therefore common ISA strings can be updated as follows to include the relevant Zc extensions, for example:

- RV32IMC becomes RV32IM_Zce
- RV32IMCF becomes RV32IMF_Zce

29.4. MISA.C

MISA.C is set if the following extensions are selected:

- Zca and not F
- Zca, Zcf and F (but not D) is specified (RV32 only)
- Zca, Zcf and Zcd if D is specified (RV32 only)
  - this configuration excludes Zcmp, Zcmt

- Zca, Zcd if D is specified (RV64 only)
  - this configuration excludes Zcmp, Zcmt

29.5. Zca

The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores.

Therefore it *excluded* all 16-bit floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*, *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.

the C extension only includes F/D instructions when D and F are also specified

29.6. Zcf (RV32 only)

Zcf is the existing set of compressed single precision floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*.

Zcf is only relevant to RV32, it cannot be specified for RV64.

The Zcf extension depends on the Zca and F extensions.

29.7. Zcd

Zcd is the existing set of compressed double precision floating point loads and stores: *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.

The Zcd extension depends on the Zca and D extensions.

29.8. Zcb

Zcb has simple code-size saving instructions which are easy to implement on all CPUs.

All encodings are currently reserved for all architectures, and have no conflicts with any existing extensions.

Zcb can be implemented on any CPU as the instructions are 16-bit versions of existing 32-bit instructions from the application class profile.

The Zcb extension depends on the Zca extension.

As shown on the individual instruction pages, many of the instructions in Zcb depend upon another extension being implemented. For example, *c.mul* is only implemented if M or Zmmul is implemented, and *c.sext.b* is only implemented if Zbb is implemented.

The *c.mul* encoding uses the CA register format along with other instructions such as *c.sub*, *c.xor* etc.

c.sext.w is a pseudoinstruction for c.addiw rd, 0 (RV64)

RV32	RV64	Mnemonic	Instruction
yes	yes	c.lbu *rd'*, uimm(*rs1'*)	Load unsigned byte, 16-bit encoding

RV32	RV64	Mnemonic	Instruction
yes	yes	c.lhu *rd'*, uimm(*rs1'*)	Load unsigned halfword, 16-bit encoding
yes	yes	c.lh *rd'*, uimm(*rs1'*)	Load signed halfword, 16-bit encoding
yes	yes	c.sb *rs2'*, uimm(*rs1'*)	Store byte, 16-bit encoding
yes	yes	c.sh *rs2'*, uimm(*rs1'*)	Store halfword, 16-bit encoding
yes	yes	c.zext.b *rsd'*	Zero extend byte, 16-bit encoding
yes	yes	c.sext.b *rsd'*	Sign extend byte, 16-bit encoding
yes	yes	c.zext.h *rsd'*	Zero extend halfword, 16-bit encoding
yes	yes	c.sext.h *rsd'*	Sign extend halfword, 16-bit encoding
	yes	c.zext.w *rsd'*	Zero extend word, 16-bit encoding
yes	yes	c.not *rsd'*	Bitwise not, 16-bit encoding
yes	yes	c.mul *rsd'*, *rs2'*	Multiply, 16-bit encoding

29.9. Zcmp

The Zcmp extension is a set of instructions which may be executed as a series of existing 32-bit RISC-V instructions.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with Zcd, which is included when C and D extensions are both present.

Zcmp is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with application class profiles.

The Zcmp extension depends on the Zca extension.

The PUSH/POP assembly syntax uses several variables, the meaning of which are:

- *reg_list* is a list containing 1 to 13 registers (ra and 0 to 12 s registers)
  - valid values: {ra}, \{ra, s0}, \{ra, s0-s1}, \{ra, s0-s2}, ..., \{ra, s0-s8}, \{ra, s0-s9}, \{ra, s0-s11}
  - note that \{ra, s0-s10} is *not* valid, giving 12 lists not 13 for better encoding
- *stack_adj* is the total size of the stack frame.
  - valid values vary with register list length and the specific encoding, see the instruction pages for details.

RV32	RV64	Mnemonic	Instruction
yes	yes	cm.push *{reg_list}, -stack_adj*	cm.push
yes	yes	cm.pop *{reg_list}, stack_adj*	cm.pop
yes	yes	cm.popret *{reg_list}, stack_adj*	cm.popret
yes	yes	cm.popretz *{reg_list}, stack_adj*	cm.popretz
yes	yes	cm.mva01s *rs1', rs2'*	Move two s0-s7 registers into a0-a1
yes	yes	cm.mvsa01 *r1s', r2s'*	Move a0-a1 into two different s0-s7 registers

29.10. Zcmt

Zcmt adds the table jump instructions and also adds the jvt CSR. The jvt CSR requires a state enable if Smstateen is implemented. See jvt CSR, table jump base vector and control register for details.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with Zcd, which is included when C and D extensions are both present.

Zcmt is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with RVA profiles.

The Zcmt extension depends on the Zca and Zicsr extensions.

RV32	RV64	Mnemonic	Instruction
yes	yes	cm.jt *index*	Jump via table
yes	yes	cm.jalt *index*	Jump and link via table

29.11. Zc instruction formats

Several instructions in this specification use the following new instruction formats.

Format	instructions	15:10	9	8	7	6	5	4	3	2	1	0
CLB	c.lbu	funct6	rs1'			uimm		rd'			op	
CSB	c.sb	funct6	rs1'			uimm		rs2'			op	
CLH	c.lhu, c.lh	funct6	rs1'			funct1	uimm	rd'			op	
CSH	c.sh	funct6	rs1'			funct1	uimm	rs2'			op	
CU	c.[sz]ext.*, c.not	funct6	rd'/rs1'			funct5					op	
CMMV	cm.mvsa01 cm.mva01s	funct6	r1s'			funct2		r2s'			op	
CMJT	cm.jt cm.jalt	funct6	index								op	
CMPP	cm.push*, cm.pop*	funct6	funct2		urlist				spimm		op	

c.mul uses the existing CA format.

29.12. Zcb instructions

29.12.1. c.lbu

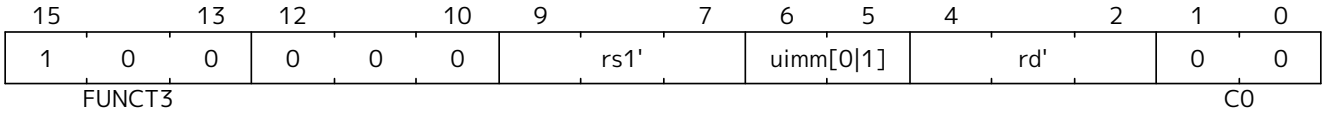
Synopsis

Load unsigned byte, 16-bit encoding

Mnemonic

c.lbu *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction loads a byte from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting byte is zero extended to XLEN bits and is written to *rd'*.

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTZ(mem[X(rs1c)+EXTZ(uimm)][7..0]);
```

29.12.2. c.lhu

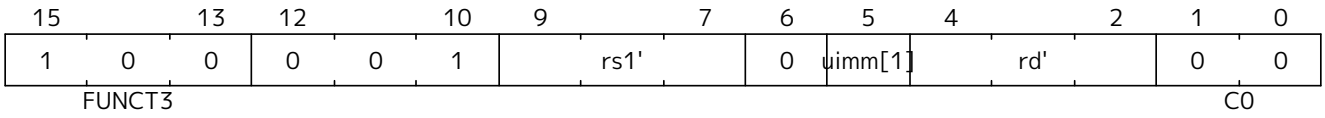
Synopsis

Load unsigned halfword, 16-bit encoding

Mnemonic

c.lhu *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting halfword is zero extended to XLEN bits and is written to *rd'*.

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTZ(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

29.12.3. c.lh

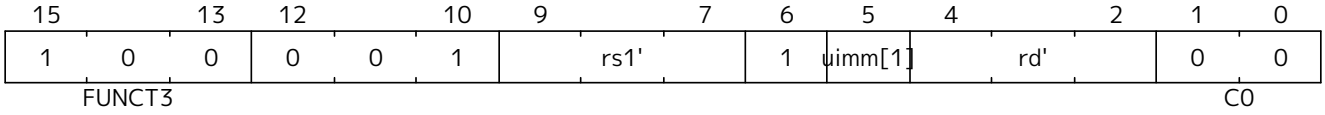
Synopsis

Load signed halfword, 16-bit encoding

Mnemonic

c.lh *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting halfword is sign extended to XLEN bits and is written to *rd'*.

rd' and rs1' are from the standard 8-register set x8-x15.

Prerequisites

None

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

X(rdc) = EXTS(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

29.12.4. c.sb

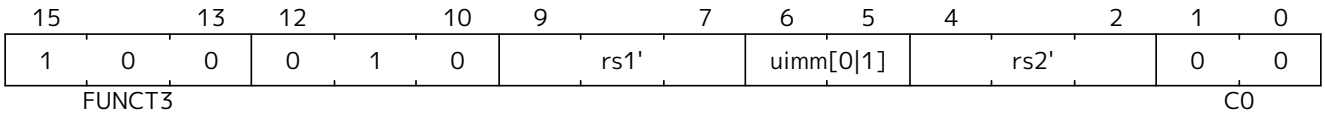
Synopsis

Store byte, 16-bit encoding

Mnemonic

c.sb *rs2'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction stores the least significant byte of *rs2'* to the memory address formed by adding *rs1'* to the zero extended immediate *uimm*.

rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

None

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][7..0] = X(rs2c)
```

29.12.5. c.sh

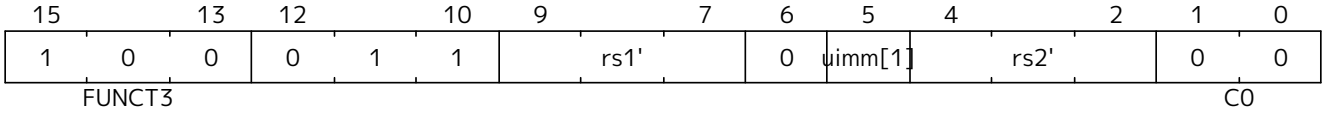
Synopsis

Store halfword, 16-bit encoding

Mnemonic

c.sh *rs2'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction stores the least significant halfword of *rs2'* to the memory address formed by adding *rs1'* to the zero extended immediate *uimm*.

rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

None

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][15..0] = X(rs2c)
```

29.12.6. c.zext.b

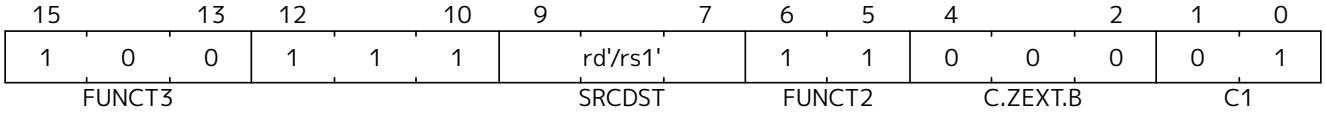
Synopsis

Zero extend byte, 16-bit encoding

Mnemonic

c.zext.b *rd'/rs1'*

Encoding (RV32, RV64):



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant byte of the operand to XLEN bits by inserting zeros into all of the bits more significant than 7.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent:

```
andi rd'/rs1', rd'/rs1', 0xff
```

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[7..0]);
```

29.12.7. c.sext.b

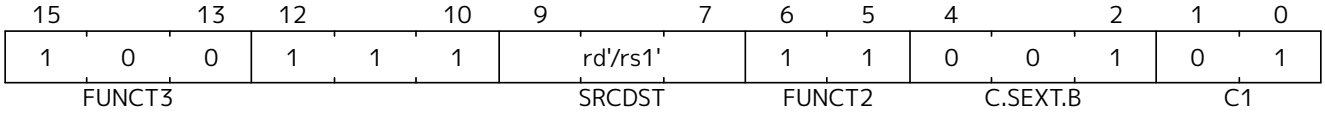
Synopsis

Sign extend byte, 16-bit encoding

Mnemonic

c.sext.b *rd'/rs1'*

Encoding (RV32, RV64):



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand to XLEN bits by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = EXTS(X(rsdc)[7..0]);
```

29.12.8. c.zext.h

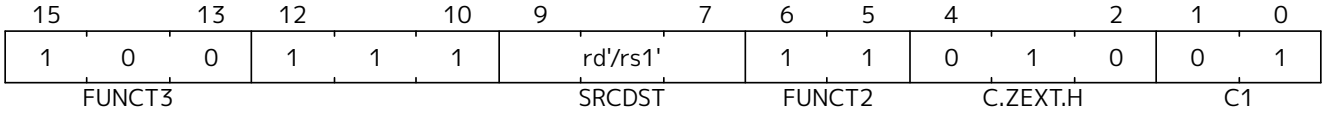
Synopsis

Zero extend halfword, 16-bit encoding

Mnemonic

c.zext.h *rd'/rs1'*

Encoding (RV32, RV64):



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant halfword of the operand to XLEN bits by inserting zeros into all of the bits more significant than 15.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[15..0]);
```

29.12.9. c.sext.h

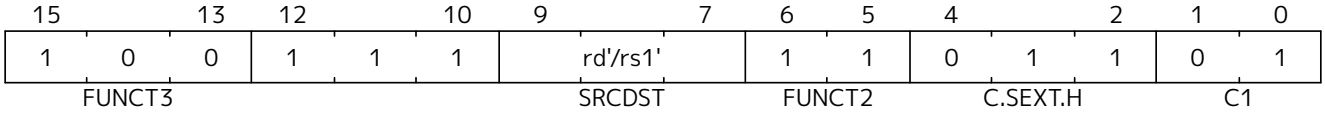
Synopsis

Sign extend halfword, 16-bit encoding

Mnemonic

c.sext.h *rd'/rs1'*

Encoding (RV32, RV64):



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant halfword in the operand to XLEN bits by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = EXTS(X(rsdc)[15..0]);
```

29.12.10. c.zext.w

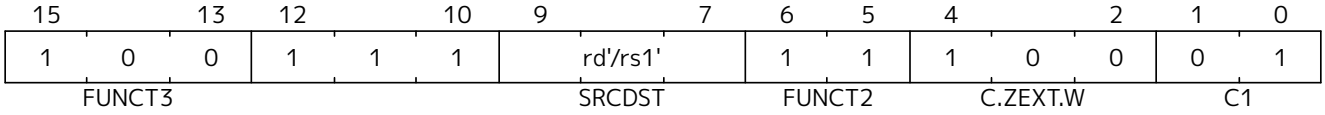
Synopsis

Zero extend word, 16-bit encoding

Mnemonic

c.zext.w *rd'/rs1'*

Encoding (RV64):



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant word of the operand to XLEN bits by inserting zeros into all of the bits more significant than 31.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

Zba is also required.

32-bit equivalent:

```
add.uw rd'/rs1', rd'/rs1', zero
```

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[31..0]);
```

29.12.11. c.not

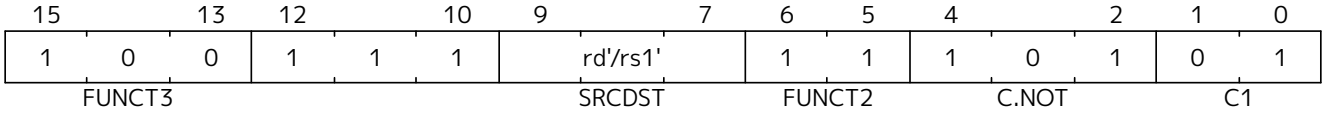
Synopsis

Bitwise not, 16-bit encoding

Mnemonic

c.not *rd'/rs1'*

Encoding (RV32, RV64):



Description

This instruction takes the one's complement of *rd'/rs1'* and writes the result to the same register.

rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent:

```
xori rd'/rs1', rd'/rs1', -1
```

The SAIL module variable for rd'/rs1' is called rsdc.

Operation

```
X(rsdc) = X(rsdc) XOR -1;
```

29.12.12. c.mul

Synopsis

Multiply, 16-bit encoding

Mnemonic

c.mul *rsd'*, *rs2'*

Encoding (RV32, RV64):

Description

This instruction multiplies XLEN bits of the source operands from *rsd'* and *rs2'* and writes the lowest XLEN bits of the result to *rsd'*.

rd'/rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites

M or Zmmul must be configured.

The SAIL module variable for rd'/rs1' is called rsdc, and for rs2' is called rs2c.

Operation

```
let result_wide = to_bits(2 * sizeof(xlen), signed(X(rsdc)) * signed(X(rs2c)));
X(rsdc) = result_wide[(sizeof(xlen) - 1) .. 0];
```

29.13. PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- cm.push
- cm.pop
- cm.popret
- cm.popretz

The term PUSH refers to *cm.push*.

The term POP refers to *cm.pop*.

The term POPRET refers to *cm.popret and cm.popretz*.

Common details for these instructions are in this section.

29.13.1. PUSH/POP functional overview

PUSH, POP, POPRET are used to reduce the size of function prologues and epilogues.

1. The PUSH instruction
   - adjusts the stack pointer to create the stack frame
   - pushes (stores) the registers specified in the register list to the stack frame
2. The POP instruction
   - pops (loads) the registers in the register list from the stack frame
   - adjusts the stack pointer to destroy the stack frame
3. The POPRET instructions
   - pop (load) the registers in the register list from the stack frame
   - *cm.popretz* also moves zero into *a0* as the return value
   - adjust the stack pointer to destroy the stack frame
   - execute a *ret* instruction to return from the function

29.13.2. Example usage

This example gives an illustration of the use of PUSH and POPRET.

The function *processMarkers* in the EMBench benchmark picojpeg in the following file on github: libpicojpeg.c

The prologue and epilogue compile with GCC10 to:

```
   0001098a <processMarkers>:
   1098a:       711d                    addi    sp,sp,-96 ;#cm.push(1)
   1098c:       c8ca                    sw      s2,80(sp) ;#cm.push(2)
   1098e:       c6ce                    sw      s3,76(sp) ;#cm.push(3)
   10990:       c4d2                    sw      s4,72(sp) ;#cm.push(4)
   10992:       ce86                    sw      ra,92(sp) ;#cm.push(5)
   10994:       cca2                    sw      s0,88(sp) ;#cm.push(6)
   10996:       caa6                    sw      s1,84(sp) ;#cm.push(7)
   10998:       c2d6                    sw      s5,68(sp) ;#cm.push(8)
   1099a:       c0da                    sw      s6,64(sp) ;#cm.push(9)
   1099c:       de5e                    sw      s7,60(sp) ;#cm.push(10)
   1099e:       dc62                    sw      s8,56(sp) ;#cm.push(11)
   109a0:       da66                    sw      s9,52(sp) ;#cm.push(12)
   109a2:       d86a                    sw      s10,48(sp);#cm.push(13)
   109a4:       d66e                    sw      s11,44(sp);#cm.push(14)
...
   109f4:       4501                    li      a0,0      ;#cm.popretz(1)
   109f6:       40f6                    lw      ra,92(sp) ;#cm.popretz(2)
   109f8:       4466                    lw      s0,88(sp) ;#cm.popretz(3)
   109fa:       44d6                    lw      s1,84(sp) ;#cm.popretz(4)
   109fc:       4946                    lw      s2,80(sp) ;#cm.popretz(5)
   109fe:       49b6                    lw      s3,76(sp) ;#cm.popretz(6)
   10a00:       4a26                    lw      s4,72(sp) ;#cm.popretz(7)
   10a02:       4a96                    lw      s5,68(sp) ;#cm.popretz(8)
   10a04:       4b06                    lw      s6,64(sp) ;#cm.popretz(9)
   10a06:       5bf2                    lw      s7,60(sp) ;#cm.popretz(10)
   10a08:       5c62                    lw      s8,56(sp) ;#cm.popretz(11)
   10a0a:       5cd2                    lw      s9,52(sp) ;#cm.popretz(12)
   10a0c:       5d42                    lw      s10,48(sp);#cm.popretz(13)
   10a0e:       5db2                    lw      s11,44(sp);#cm.popretz(14)
   10a10:       6125                    addi    sp,sp,96  ;#cm.popretz(15)
   10a12:       8082                    ret               ;#cm.popretz(16)
```

with the GCC option *-msave-restore* the output is the following:

```
0001080e <processMarkers>:
   1080e:       73a012ef                jal     t0,11f48 <__riscv_save_12>
   10812:       1101                    addi    sp,sp,-32
...
   10862:       4501                    li      a0,0
   10864:       6105                    addi    sp,sp,32
   10866:       71e0106f                j       11f84 <__riscv_restore_12>
```

with PUSH/POPRET this reduces to

```
0001080e <processMarkers>:
   1080e:       b8fa                    cm.push    \{ra,s0-s11},-96
...
   10866:       bcfa                    cm.popretz \{ra,s0-s11}, 96
```

The prologue / epilogue reduce from 60-bytes in the original code, to 14-bytes with *-msave-restore*, and to 4-bytes with PUSH and POPRET. As well as reducing the code-size PUSH and POPRET eliminate the branches from calling the millicode *save/restore* routines and so may also perform better.

The calls to <riscv_save_0>/<riscv_restore_0> become 64-bit when the target functions are out of the ±1 MB range, increasing the prologue/epilogue size to 22-bytes.

POP is typically used in tail-calling sequences where ret is not used to return to ra after destroying the stack frame.

29.13.2.1. Stack pointer adjustment handling

The instructions all automatically adjust the stack pointer by enough to cover the memory required for the registers being saved or restored. Additionally the *spimm* field in the encoding allows the stack pointer to be adjusted in additional increments of 16-bytes. There is only a small restricted range available in the encoding; if the range is insufficient then a separate *c.addi16sp* can be used to increase the range.

29.13.2.2. Register list handling

There is no support for the *\{ra, s0-s10}* register list without also adding *s11*. Therefore the *\{ra, s0-s11}* register list must be used in this case.

29.13.3. PUSH/POP Fault handling

Correct execution requires that *sp* refers to idempotent memory (also see Non-idempotent memory handling), because the core must be able to handle traps detected during the sequence. The entire PUSH/POP sequence is re-executed after returning from the trap handler, and multiple traps are possible during the sequence.

If a trap occurs during the sequence then *xEPC* is updated with the PC of the instruction, *xTVAL* (if not read-only-zero) updated with the bad address if it was an access fault and *xCAUSE* updated with the type of trap.

It is implementation defined whether interrupts can also be taken during the sequence execution.

29.13.4. Software view of execution

29.13.4.1. Software view of the PUSH sequence

From a software perspective the PUSH sequence appears as:

- A sequence of stores writing the bytes required by the pseudocode
  - The bytes may be written in any order.
  - The bytes may be grouped into larger accesses.
  - Any of the bytes may be written multiple times.
- A stack pointer adjustment

If an implementation allows interrupts during the sequence, and the interrupt handler uses sp to allocate stack memory, then any stores which were executed before the interrupt may be overwritten by the handler. This is safe because the memory is idempotent and the stores will be re-executed when execution resumes.

The stack pointer adjustment must only be committed only when it is certain that the entire PUSH instruction will commit.

Stores may also return imprecise faults from the bus. It is platform defined whether the core implementation waits for the bus responses before continuing to the final stage of the sequence, or handles errors responses after completing the PUSH instruction.

For example:

```
cm.push  \{ra, s0-s5}, -64
```

Appears to software as:

```
# any bytes from sp-1 to sp-28 may be written multiple times before
# the instruction completes therefore these updates may be visible in
# the interrupt/exception handler below the stack pointer
sw  s5, -4(sp)
sw  s4, -8(sp)
sw  s3,-12(sp)
sw  s2,-16(sp)
sw  s1,-20(sp)
sw  s0,-24(sp)
sw  ra,-28(sp)

# this must only execute once, and will only execute after all stores
# completed without any precise faults, therefore this update is only
# visible in the interrupt/exception handler if cm.push has completed
addi sp, sp, -64
```

29.13.4.2. Software view of the POP/POPRET sequence

From a software perspective the POP/POPRET sequence appears as:

- A sequence of loads reading the bytes required by the pseudocode.
  - The bytes may be loaded in any order.
  - The bytes may be grouped into larger accesses.
  - Any of the bytes may be loaded multiple times.
- A stack pointer adjustment
- An optional `li a0, 0`
- An optional `ret`

If a trap occurs during the sequence, then any loads which were executed before the trap may update architectural state. The loads will be re-executed once the trap handler completes, so the values will be overwritten. Therefore it is permitted for an implementation to update some of the destination registers before taking a fault.

The optional `li a0, 0`, stack pointer adjustment and optional `ret` must only be committed only when it is certain that the entire POP/POPRET instruction will commit.

For POPRET once the stack pointer adjustment has been committed the `ret` must execute.

For example:

```
cm.popretz \{ra, s0-s3}, 32;
```

Appears to software as:

```
# any or all of these load instructions may execute multiple times
# therefore these updates may be visible in the interrupt/exception handler
lw   s3, 28(sp)
lw   s2, 24(sp)
lw   s1, 20(sp)
lw   s0, 16(sp)
lw   ra, 12(sp)

# these must only execute once, will only execute after all loads
# complete successfully all instructions must execute atomically
# therefore these updates are not visible in the interrupt/exception handler
li a0, 0
addi sp, sp, 32
ret
```

29.13.5. Non-idempotent memory handling

An implementation may have a requirement to issue a PUSH/POP instruction to non-idempotent memory.

If the core implementation does not support PUSH/POP to non-idempotent memories, the core may use an idempotency PMA to detect it and take a load (POP/POPRET) or store (PUSH) access-fault exception in order to avoid unpredictable results.

Software should only use these instructions on non-idempotent memory regions when software can tolerate the required memory accesses being issued repeatedly in the case that they cause exceptions.

29.13.6. Example RV32I PUSH/POP sequences

The examples are included show the load/store series expansion and the stack adjustment. Examples of *cm.popret* and *cm.popretz* are not included, as the difference in the expanded sequence from *cm.pop* is trivial in all cases.

29.13.6.1. cm.push \{ra, s0-s2}, -64

Encoding: *rlist*=7, *spimm*=3

expands to:

```
sw  s2,  -4(sp);
sw  s1,  -8(sp);
sw  s0, -12(sp);
sw  ra, -16(sp);
addi sp, sp, -64;
```

29.13.6.2. cm.push \{ra, s0-s11}, -112

Encoding: *rlist*=15, *spimm*=3

expands to:

```
sw  s11,  -4(sp);
sw  s10,  -8(sp);
sw  s9,  -12(sp);
sw  s8,  -16(sp);
sw  s7,  -20(sp);
sw  s6,  -24(sp);
sw  s5,  -28(sp);
sw  s4,  -32(sp);
sw  s3,  -36(sp);
sw  s2,  -40(sp);
sw  s1,  -44(sp);
sw  s0,  -48(sp);
sw  ra,  -52(sp);
addi sp, sp, -112;
```

29.13.6.3. cm.pop {ra}, 16

Encoding: *rlist*=4, *spimm*=0

expands to:

```
lw   ra, 12(sp);
addi sp, sp, 16;
```

29.13.6.4. cm.pop \{ra, s0-s3}, 48

Encoding: *rlist*=8, *spimm*=1

expands to:

```
lw   s3, 44(sp);
lw   s2, 40(sp);
lw   s1, 36(sp);
lw   s0, 32(sp);
lw   ra, 28(sp);
addi sp, sp, 48;
```

29.13.6.5. cm.pop \{ra, s0-s4}, 64

Encoding: *rlist*=9, *spimm*=2

expands to:

```
lw   s4, 60(sp);
lw   s3, 56(sp);
lw   s2, 52(sp);
lw   s1, 48(sp);
lw   s0, 44(sp);
lw   ra, 40(sp);
addi sp, sp, 64;
```

29.13.7. cm.push

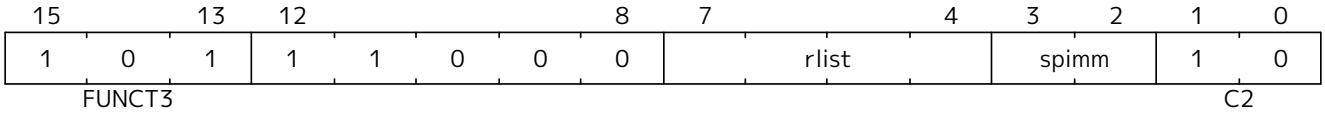
Synopsis

Create stack frame: store ra and 0 to 12 saved registers to the stack frame, optionally allocate additional stack space.

Mnemonic

cm.push *{reg_list}, -stack_adj*

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.push.e

Assembly Syntax:

```
cm.push \{reg_list},  -stack_adj
cm.push {xreg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:
switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: \{reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: \{reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: \{reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: \{reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: \{reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: \{reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: \{reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: \{reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
```

```
  case 15: \{reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
```

```
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pushes (stores) the registers in *reg_list* to the memory below the stack pointer, and then creates the stack frame by decrementing the stack pointer by *stack_adj*, including any additional stack space requested by the value of *spimm*.

All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists

Operation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("sw x[i], 0(addr)");
      8:  asm("sd x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
```

```
sp-=stack_adj;
```

29.13.8. cm.pop

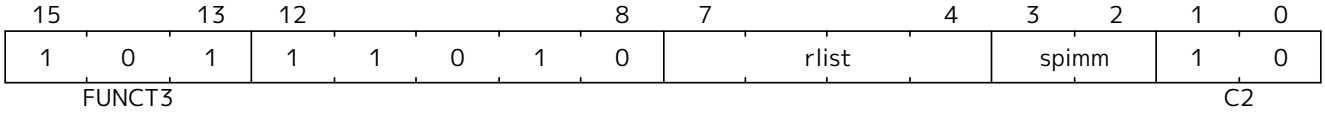
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame.

Mnemonic

cm.pop *{reg_list}, stack_adj*

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.pop.e

Assembly Syntax:

```
cm.pop \{reg_list},  stack_adj
cm.pop {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:
switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:
switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: \{reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: \{reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: \{reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: \{reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: \{reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: \{reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: \{reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: \{reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: \{reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
```

```
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
```

```
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*.

All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists

Operation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;
```

29.13.9. cm.popretz

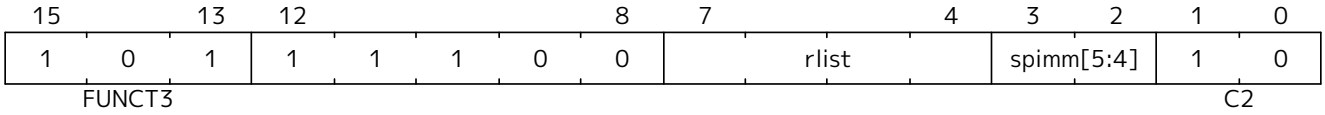
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, move zero into a0, return to ra.

Mnemonic

cm.popretz *{reg_list}, stack_adj*

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.popretz.e

Assembly Syntax:

```
cm.popretz \{reg_list},  stack_adj
cm.popretz {xreg_list}, stack_adj
```

```
RV32E:
switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: \{reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: \{reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: \{reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: \{reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: \{reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: \{reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: \{reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: \{reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: \{reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
```

```
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
```

```
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into a0 and then returns to *ra*.

All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists

Operation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

The li a0, 0 **could** *be executed more than once, but is included in the atomic section for convenience.*

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

asm("li a0, 0");
sp+=stack_adj;
asm("ret");
```

29.13.10. cm.popret

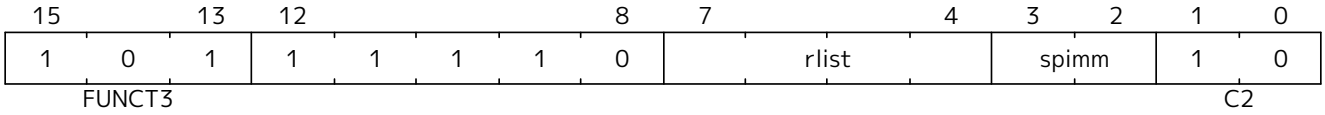
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, return to ra.

Mnemonic

cm.popret *{reg_list}, stack_adj*

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.popret.e

Assembly Syntax:

```
cm.popret \{reg_list},  stack_adj
cm.popret {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: \{reg_list="ra";         xreg_list="x1";}
  case  5: \{reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: \{reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: \{reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: \{reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: \{reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: \{reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: \{reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: \{reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: \{reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: \{reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
```

```
  //note - to include s10, s11 must also be included
  case 15: \{reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
```

```
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj* and then returns to *ra*.

All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists

Operation

The first section of pseudocode may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudocode executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;
asm("ret");
```

29.13.11. cm.mvsa01

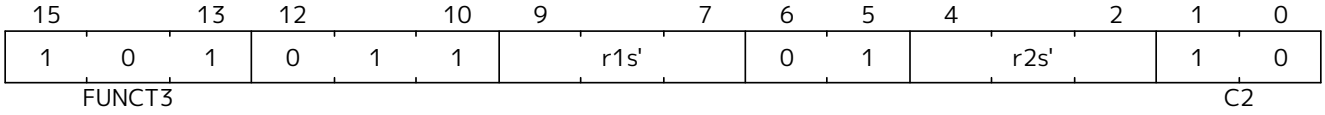
Synopsis

Move a0-a1 into two registers of s0-s7

Mnemonic

cm.mvsa01 *r1s'*, *r2s'*

Encoding (RV32, RV64):



For the encoding to be legal r1s' != r2s'.

Assembly Syntax:

```
cm.mvsa01 r1s', r2s'
```

Description

This instruction moves *a0* into *r1s'* and *a1* into *r2s'*. *r1s'* and *r2s'* must be different. The execution is atomic, so it is not possible to observe state where only one of *r1s'* or *r2s'* has been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudocode below.

The s register mapping is taken from the UABI, and may not match the currently unratified EABI. cm.mvsa01.e may be included in the future.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

29.13.12. cm.mva01s

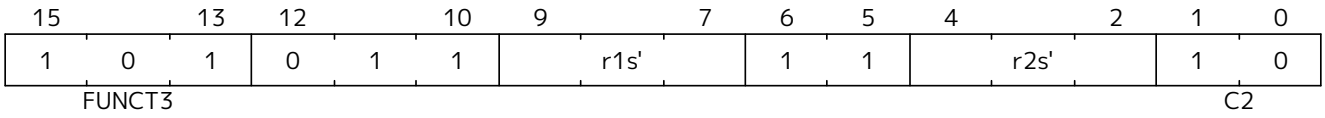
Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

cm.mva01s *r1s'*, *r2s'*

Encoding (RV32, RV64):



Assembly Syntax:

```
cm.mva01s r1s', r2s'
```

Description

This instruction moves *r1s'* into *a0* and *r2s'* into *a1*. The execution is atomic, so it is not possible to observe state where only one of *a0* or *a1* have been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudocode below.

The s register mapping is taken from the UABI, and may not match the currently unratified EABI. cm.mva01s.e may be included in the future.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

29.14. Table Jump Overview

cm.jt (Jump via table) and *cm.jalt* (Jump and link via table) are referred to as table jump.

Table jump uses a 256-entry XLEN wide table in instruction memory to contain function addresses. The table must be a minimum of 64-byte aligned.

Table entries follow the current data endianness. This is different from normal instruction fetch which is always little-endian.

cm.jt and *cm.jalt* encodings index the table, giving access to functions within the full XLEN wide address space.

This is used as a form of dictionary compression to reduce the code size of *jal* / *auipc+jalr* / *jr* / *auipc+jr* instructions.

Table jump allows the linker to replace the following instruction sequences with a *cm.jt* or *cm.jalt* encoding, and an entry in the table:

- 32-bit *j* calls
- 32-bit *jal* ra calls
- 64-bit *auipc+jr* calls to fixed locations
- 64-bit *auipc+jalr ra* calls to fixed locations
  - The *auipc+jr/jalr* sequence is used because the offset from the PC is out of the ±1 MB range.

If a return address stack is implemented, then as *cm.jalt* is equivalent to *jal ra*, it pushes to the stack.

29.14.1. jvt

The base of the table is in the jvt CSR (see jvt CSR, table jump base vector and control register), each table entry is XLEN bits.

If the same function is called with and without linking then it must have two entries in the table. This is typically caused by the same function being called with and without tail calling.

29.14.2. Table Jump Fault handling

For a table jump instruction, the table entry that the instruction selects is considered an extension of the instruction itself. Hence, the execution of a table jump instruction involves two instruction fetches, the first to read the instruction (*cm.jt*/*cm.jalt*) and the second to read from the jump vector table (JVT). Both instruction fetches are *implicit* reads, and both require execute permission; read permission is irrelevant. It is recommended that the second fetch be ignored for hardware triggers and breakpoints.

Memory writes to the jump vector table require an instruction barrier (*fence.i*) to guarantee that they are visible to the instruction fetch.

Multiple contexts may have different jump vector tables. JVT may be switched between them without an instruction barrier if the tables have not been updated in memory since the last *fence.i*.

If an exception occurs on either instruction fetch, xEPC is set to the PC of the table jump instruction, xCAUSE is set as expected for the type of fault and xTVAL (if not set to zero) contains the fetch address which caused the fault.

29.14.3. jvt CSR

Synopsis

Table jump base vector and control register

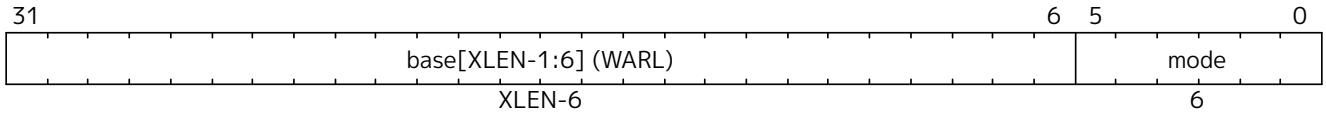
Address:

0x017

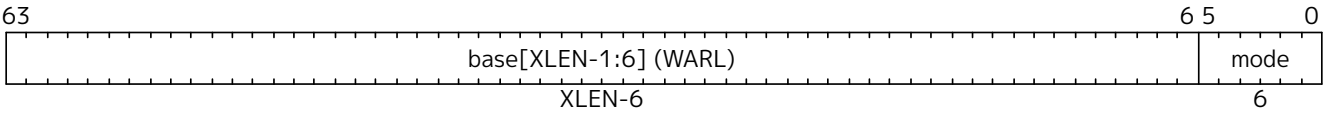
Permissions:

URW

Format (RV32):



Format (RV64):



Description

The *jvt* register is an XLEN-bit **WARL** read/write register that holds the jump table configuration, consisting of the jump table base address (BASE) and the jump table mode (MODE).

If Section 29.10 is implemented then *jvt* must also be implemented, but can contain a read-only value. If *jvt* is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 64-byte boundary. Note that the CSR contains only bits XLEN-1 through 6 of the address *base*. When computing jump-table accesses, the lower six bits of *base* are filled with zeroes to obtain an XLEN-bit jump-table base address *jvt.base* that is always aligned on a 64-byte boundary.

jvt.base is a virtual address, whenever virtual memory is enabled.

The memory pointed to by *jvt.base* is treated as instruction memory for the purpose of executing table jump instructions, implying execute access permission.

Table 41. jvt.mode definition

jvt.mode	Comment
000000	Jump table mode
others	**reserved for future standard use**

jvt.mode is a **WARL** field, so can only be programmed to modes which are implemented. Therefore the discovery mechanism is to attempt to program different modes and read back the values to see which are available. Jump table mode *must* be implemented.

in future the RISC-V Unified Discovery method will report the available modes.

Architectural State:

jvt CSR adds architectural state to the system software context (such as an OS process), therefore must be

saved/restored on context switches. <<<

29.14.4. cm.jt

Synopsis

jump via table

Mnemonic

cm.jt *index*

Encoding (RV32, RV64):

For this encoding to decode as cm.jt, index<32, otherwise it decodes as cm.jalt, see Jump and link via table.

If jvt.mode = 0 (Jump Table Mode) then cm.jt behaves as specified here. If jvt.mode is a reserved value, then cm.jt is also reserved. In the future other defined values of jvt.mode may change the behaviour of cm.jt.

Assembly Syntax:

```
cm.jt index
```

Description

cm.jt reads an entry from the jump vector table in memory and jumps to the address that was read.

For further information see Table Jump Overview.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

# table_address is temporary internal state, it doesn't represent a real
register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = jvt.base + (index<<2);
  64:  table_address[XLEN-1:0] = jvt.base + (index<<3);
}

//fetch from the jump table
pc = InstMemory[table_address][XLEN-1:0]&~0x1;  // Clear bit 0.
```

29.14.5. cm.jalt

Synopsis

jump via table with optional link

Mnemonic

cm.jalt *index*

Encoding (RV32, RV64):

For this encoding to decode as cm.jalt, index>=32, otherwise it decodes as cm.jt, see Jump via table.

If jvt.mode = 0 (Jump Table Mode) then cm.jalt behaves as specified here. If jvt.mode is a reserved value, then cm.jalt is also reserved. In the future other defined values of jvt.mode may change the behaviour of cm.jalt.

Assembly Syntax:

```
cm.jalt index
```

Description

cm.jalt reads an entry from the jump vector table in memory and jumps to the address that was read, linking to *ra*.

For further information see Table Jump Overview.

Prerequisites

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudocode. The SAIL hasn't been written yet.

# table_address is temporary internal state, it doesn't represent a real
register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = jvt.base + (index<<2);
  64:  table_address[XLEN-1:0] = jvt.base + (index<<3);
}

//fetch from the jump table

ra = pc+2;
pc = InstMemory[table_address][XLEN-1:0]&~0x1;  // Clear bit 0.
```

Chapter 30. Bit Manipulation Extensions

The bit-manipulation (bitmanip) extension collection is comprised of several component extensions to the base RISC-V architecture that are intended to provide some combination of code-size reduction, performance improvement, and energy reduction. While the instructions are intended for general use, some instructions are more useful in certain domains than in others. Hence, several smaller bitmanip extensions are provided. Each of these smaller extensions is grouped by common function and use case, and each has its own Zb*-extension name.

Each bitmanip extension includes a group of several bitmanip instructions that have similar purposes and can often share the same logic. Some instructions are available in only one extension, while others are available in several. The instructions have mnemonics and encodings that are independent of the extensions in which they appear. Thus, when implementing extensions with overlapping instructions, there is no redundancy in logic or encoding.

The bitmanip extensions are defined for RV32 and RV64.

The bitmanip extension follows the convention in RV64 that *w*-suffixed instructions (without a dot before the *w*) ignore the upper 32 bits of their inputs, operate on the least-significant 32 bits as signed values, and produce a 32-bit signed result that is sign-extended to XLEN.

Bitmanip instructions with the suffix *.uw* have one operand that is an unsigned 32-bit value that is extracted from the least-significant 32 bits of the specified register. Other than that, these perform full-XLEN operations.

Bitmanip instructions with the suffixes *.b*, *.h*, and *.w* only look at the least-significant 8 bits, 16 bits, and 32 bits of the input (respectively) and produce an XLEN-wide result that is sign-extended or zero-extended, based on the specific instruction.

The bit-manipulation instructions comprise the following extensions:

- Zba: Address generation instructions
- Zbb: Basic bit-manipulation
- Zbs: Single-bit instructions
- Zbc: Carry-less multiplication
- Zbkb: Bit-manipulation for Cryptography
- Zbkc: Carry-less multiplication for Cryptography
- Zbkx: Crossbar permutations

Below is a list of all of the instructions that are included in these extensions, along with their specific mapping:

RV32	RV64	Mnemonic	Instruction	Zbb	Zbkb	Zbc	Zbkc
✓	✓	andn *rd*, *rs1*, *rs2*	AND with inverted operand	✓	✓		
✓	✓	brev8 *rd*, *rs*	Reverse bits in bytes		✓		
✓	✓	clmul *rd*, *rs1*, *rs2*	Carry-less multiply (low-part)			✓	✓
✓	✓	clmulh *rd*, *rs1*, *rs2*	Carry-less multiply (high-part)			✓	✓
✓	✓	clmulr *rd*, *rs1*, *rs2*	Carry-less multiply (reversed)			✓	
✓	✓	clz *rd*, *rs*	Count leading zero bits	✓			

RV32	RV64	Mnemonic	Instruction	Zbb	Zbkb	Zbc	Zbkc
	✓	clzw *rd*, *rs*	Count leading zero bits in word	✓			
✓	✓	cpop *rd*, *rs*	Count set bits	✓			
	✓	cpopw *rd*, *rs*	Count set bits in word	✓			
✓	✓	ctz *rd*, *rs*	Count trailing zero bits	✓			
	✓	ctzw *rd*, *rs*	Count trailing zero bits in word	✓			
✓	✓	max *rd*, *rs1*, *rs2*	Maximum	✓			
✓	✓	maxu *rd*, *rs1*, *rs2*	Unsigned maximum	✓			
✓	✓	min *rd*, *rs1*, *rs2*	Minimum	✓			
✓	✓	minu *rd*, *rs1*, *rs2*	Unsigned minimum	✓			
✓	✓	orc.b *rd*, *rs*	Bitwise OR-Combine, byte granule	✓			
✓	✓	orn *rd*, *rs1*, *rs2*	OR with inverted operand	✓	✓		
✓	✓	pack *rd*, *rs1*, *rs2*	Pack low halves of registers		✓		
✓	✓	packh *rd*, *rs1*, *rs2*	Pack low bytes of registers		✓		
	✓	packw *rd*, *rs1*, *rs2*	Pack low 16-bits of registers (RV64)		✓		
✓	✓	rev8 *rd*, *rs*	Byte-reverse register	✓	✓		
✓	✓	rol *rd*, *rs1*, *rs2*	Rotate left (Register)	✓	✓		
	✓	rolw *rd*, *rs1*, *rs2*	Rotate Left Word (Register)	✓	✓		
✓	✓	ror *rd*, *rs1*, *rs2*	Rotate right (Register)	✓	✓		
✓	✓	rori *rd*, *rs1*, *shamt*	Rotate right (Immediate)	✓	✓		
	✓	roriw *rd*, *rs1*, *shamt*	Rotate right Word (Immediate)	✓	✓		
	✓	rorw *rd*, *rs1*, *rs2*	Rotate right Word (Register)	✓	✓		
✓	✓	sext.b *rd*, *rs*	Sign-extend byte	✓			
✓	✓	sext.h *rd*, *rs*	Sign-extend halfword	✓			
✓		unzip *rd*, *rs*	Bit deinterleave		✓		
✓	✓	xnor *rd*, *rs1*, *rs2*	Exclusive NOR	✓	✓		
✓	✓	zext.h *rd*, *rs*	Zero-extend halfword	✓			
✓		zip *rd*, *rs*	Bit interleave		✓		

RV32	RV64	Mnemonic	Instruction	Zba	Zbs
	✓	add.uw *rd*, *rs1*, *rs2*	Add unsigned word	✓	
✓	✓	bclr *rd*, *rs1*, *rs2*	Single-Bit Clear (Register)		✓
✓	✓	bclri *rd*, *rs1*, *imm*	Single-Bit Clear (Immediate)		✓
✓	✓	bext *rd*, *rs1*, *rs2*	Single-Bit Extract (Register)		✓
✓	✓	bexti *rd*, *rs1*, *imm*	Single-Bit Extract (Immediate)		✓
✓	✓	binv *rd*, *rs1*, *rs2*	Single-Bit Invert (Register)		✓
✓	✓	binvi *rd*, *rs1*, *imm*	Single-Bit Invert (Immediate)		✓
✓	✓	bset *rd*, *rs1*, *rs2*	Single-Bit Set (Register)		✓
✓	✓	bseti *rd*, *rs1*, *imm*	Single-Bit Set (Immediate)		✓
✓	✓	sh1add *rd*, *rs1*, *rs2*	Shift left by 1 and add	✓	

RV32	RV64	Mnemonic	Instruction	Zba	Zbs
	✓	sh1add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 1 and add	✓	
✓	✓	sh2add *rd*, *rs1*, *rs2*	Shift left by 2 and add	✓	
	✓	sh2add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 2 and add	✓	
✓	✓	sh3add *rd*, *rs1*, *rs2*	Shift left by 3 and add	✓	
	✓	sh3add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 3 and add	✓	
	✓	slli.uw *rd*, *rs1*, *imm*	Shift-left unsigned word (Immediate)	✓	

30.1. "B" Extension for Bit Manipulation, Version 1.0.0

The B standard extension comprises instructions provided by the Zba, Zbb, and Zbs extensions.

30.2. Zba: Extension for Address generation, Version 1.0.0

The Zba instructions can be used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and XLEN-sized indices: a shifted index is added to a base address.

The shift and add instructions do a left shift of 1, 2, or 3 because these are commonly found in real-world code and because they can be implemented with a minimal amount of additional hardware beyond that of the simple adder. This avoids lengthening the critical path in implementations.

While the shift and add instructions are limited to a maximum left shift of 3, the slli instruction (from the base ISA) can be used to perform similar shifts for indexing into arrays of wider elements. The slli.uw — added in this extension — can be used when the index is to be interpreted as an unsigned word.

The following instructions comprise the Zba extension:

RV32	RV64	Mnemonic	Instruction
	✓	add.uw *rd*, *rs1*, *rs2*	Add unsigned word
✓	✓	sh1add *rd*, *rs1*, *rs2*	Shift left by 1 and add
	✓	sh1add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 1 and add
✓	✓	sh2add *rd*, *rs1*, *rs2*	Shift left by 2 and add
	✓	sh2add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 2 and add
✓	✓	sh3add *rd*, *rs1*, *rs2*	Shift left by 3 and add
	✓	sh3add.uw *rd*, *rs1*, *rs2*	Shift unsigned word left by 3 and add
	✓	slli.uw *rd*, *rs1*, *imm*	Shift-left unsigned word (Immediate)

30.3. Zbb: Extension for Basic bit-manipulation, Version 1.0.0

30.3.1. Logical with negate

RV32	RV64	Mnemonic	Instruction
✓	✓	andn *rd*, *rs1*, *rs2*	AND with inverted operand
✓	✓	orn *rd*, *rs1*, *rs2*	OR with inverted operand

RV32	RV64	Mnemonic	Instruction
✓	✓	xnor *rd*, *rs1*, *rs2*	Exclusive NOR

Implementation Hint

The Logical with Negate instructions can be implemented by inverting the rs2 inputs to the base-required AND, OR, and XOR logic instructions. In some implementations, the inverter on rs2 used for subtraction can be reused for this purpose.

30.3.2. Count leading/trailing zero bits

RV32	RV64	Mnemonic	Instruction
✓	✓	clz *rd*, *rs*	Count leading zero bits
	✓	clzw *rd*, *rs*	Count leading zero bits in word
✓	✓	ctz *rd*, *rs*	Count trailing zero bits
	✓	ctzw *rd*, *rs*	Count trailing zero bits in word

30.3.3. Count population

These instructions count the number of set bits (1-bits). This is also commonly referred to as population count.

RV32	RV64	Mnemonic	Instruction
✓	✓	cpop *rd*, *rs*	Count set bits
	✓	cpopw *rd*, *rs*	Count set bits in word

30.3.4. Integer minimum/maximum

The integer minimum/maximum instructions are arithmetic R-type instructions that return the smaller/larger of two operands.

RV32	RV64	Mnemonic	Instruction
✓	✓	max *rd*, *rs1*, *rs2*	Maximum
✓	✓	maxu *rd*, *rs1*, *rs2*	Unsigned maximum
✓	✓	min *rd*, *rs1*, *rs2*	Minimum
✓	✓	minu *rd*, *rs1*, *rs2*	Unsigned minimum

30.3.5. Sign extension and zero extension

These instructions perform the sign extension or zero extension of the least-significant 8 bits or 16 bits of the source register.

These instructions replace the generalized idioms `slli rd,rs,(XLEN-<size>) + srai` (for sign extension of 8-bit and 16-bit quantities) and `slli + srli` (for zero extension of 16-bit quantities).

RV32	RV64	Mnemonic	Instruction
✓	✓	sext.b *rd*, *rs*	Sign-extend byte

RV32	RV64	Mnemonic	Instruction
✓	✓	sext.h *rd*, *rs*	Sign-extend halfword
✓	✓	zext.h *rd*, *rs*	Zero-extend halfword

30.3.6. Bitwise rotation

Bitwise rotation instructions are similar to the shift-logical operations from the base spec. However, where the shift-logical instructions shift in zeros, the rotate instructions shift in the bits that were shifted out of the other side of the value. Such operations are also referred to as 'circular shifts'.

RV32	RV64	Mnemonic	Instruction
✓	✓	rol *rd*, *rs1*, *rs2*	Rotate left (Register)
	✓	rolw *rd*, *rs1*, *rs2*	Rotate Left Word (Register)
✓	✓	ror *rd*, *rs1*, *rs2*	Rotate right (Register)
✓	✓	rori *rd*, *rs1*, *shamt*	Rotate right (Immediate)
	✓	roriw *rd*, *rs1*, *shamt*	Rotate right Word (Immediate)
	✓	rorw *rd*, *rs1*, *rs2*	Rotate right Word (Register)

Architecture Explanation

The rotate instructions were included to replace a common four-instruction sequence to achieve the same effect (neg; sll/srl; srl/sll; or)

30.3.7. OR Combine

orc.b sets the bits of each byte in the result *rd* to all zeros if no bit within the respective byte of *rs* is set, or to all ones if any bit within the respective byte of *rs* is set.

One use-case is string-processing functions, such as **strlen** and **strcpy**, which can use **orc.b** to test for the terminating zero byte by counting the set bits in leading non-zero bytes in a word.

RV32	RV64	Mnemonic	Instruction
✓	✓	orc.b *rd*, *rs*	Bitwise OR-Combine, byte granule

30.3.8. Byte-reverse

rev8 reverses the byte-ordering of *rs*.

RV32	RV64	Mnemonic	Instruction
✓	✓	rev8 *rd*, *rs*	Byte-reverse register

30.4. Zbc: Extension for Carry-less multiplication, Version 1.0.0

Carry-less multiplication is the multiplication in the polynomial ring over GF(2).

clmul produces the lower half of the carry-less product and **clmulh** produces the upper half of the 2×XLEN carry-less product.

clmulr produces bits 2×XLEN−2:XLEN-1 of the 2×XLEN carry-less product.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul *rd*, *rs1*, *rs2*	Carry-less multiply (low-part)
✓	✓	clmulh *rd*, *rs1*, *rs2*	Carry-less multiply (high-part)
✓	✓	clmulr *rd*, *rs1*, *rs2*	Carry-less multiply (reversed)

30.5. Zbs: Extension for Single-bit instructions, Version 1.0.0

The single-bit instructions provide a mechanism to set, clear, invert, or extract a single bit in a register. The bit is specified by its index.

RV32	RV64	Mnemonic	Instruction
✓	✓	bclr *rd*, *rs1*, *rs2*	Single-Bit Clear (Register)
✓	✓	bclri *rd*, *rs1*, *imm*	Single-Bit Clear (Immediate)
✓	✓	bext *rd*, *rs1*, *rs2*	Single-Bit Extract (Register)
✓	✓	bexti *rd*, *rs1*, *imm*	Single-Bit Extract (Immediate)
✓	✓	binv *rd*, *rs1*, *rs2*	Single-Bit Invert (Register)
✓	✓	binvi *rd*, *rs1*, *imm*	Single-Bit Invert (Immediate)
✓	✓	bset *rd*, *rs1*, *rs2*	Single-Bit Set (Register)
✓	✓	bseti *rd*, *rs1*, *imm*	Single-Bit Set (Immediate)

30.6. Zbkb: Extension for Bit-manipulation for Cryptography, Version 1.0.0

This extension contains instructions essential for implementing common operations in cryptographic workloads.

RV32	RV64	Mnemonic	Instruction
✓	✓	rol	Rotate left (Register)
	✓	rolw	Rotate Left Word (Register)
✓	✓	ror	Rotate right (Register)
✓	✓	rori	Rotate right (Immediate)
	✓	roriw	Rotate right Word (Immediate)
	✓	rorw	Rotate right Word (Register)
✓	✓	andn	AND with inverted operand
✓	✓	orn	OR with inverted operand
✓	✓	xnor	Exclusive NOR
✓	✓	pack	Pack low halves of registers
✓	✓	packh	Pack low bytes of registers
	✓	packw	Pack low 16-bits of registers (RV64)
✓	✓	brev8	Reverse bits in bytes
✓	✓	rev8	Byte-reverse register
✓		zip	Bit interleave
✓		unzip	Bit deinterleave

30.7. Zbkc: Extension for Carry-less multiplication for Cryptography, Version 1.0.0

Carry-less multiplication is the multiplication in the polynomial ring over GF(2). This is a critical operation in some cryptographic workloads, particularly the AES-GCM authenticated encryption scheme. This extension provides only the instructions needed to efficiently implement the GHASH operation, which is part of this workload.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul *rd*, *rs1*, *rs2*	Carry-less multiply (low-part)
✓	✓	clmulh *rd*, *rs1*, *rs2*	Carry-less multiply (high-part)

30.8. Zbkx: Extension for Crossbar permutations, Version 1.0.0

These instructions implement a "lookup table" for 4 and 8 bit elements inside the general purpose registers. *rs1* is used as a vector of N-bit words, and *rs2* as a vector of N-bit indices into *rs1*. Elements in *rs1* are replaced by the indexed element in *rs2*, or zero if the index into *rs2* is out of bounds.

These instructions are useful for expressing N-bit to N-bit boolean operations, and implementing cryptographic code with secret dependent memory accesses (particularly SBoxes) such that the execution latency does not depend on the (secret) data being operated on.

RV32	RV64	Mnemonic	Instruction
✓	✓	xperm4 *rd*, *rs1*, *rs2*	Crossbar permutation (nibbles)
✓	✓	xperm8 *rd*, *rs1*, *rs2*	Crossbar permutation (bytes)

30.9. Instructions (in alphabetical order)

The semantics of each instruction is expressed in a SAIL-like syntax.

30.9.1. add.uw

Synopsis

Add unsigned word

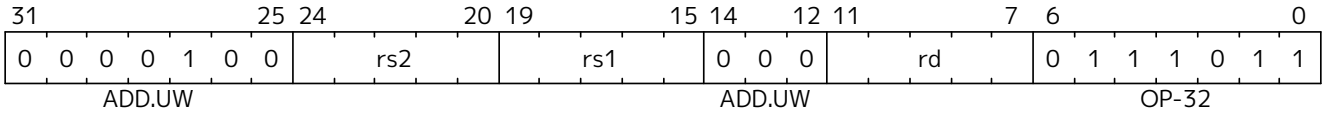
Mnemonic

add.uw *rd*, *rs1*, *rs2*

Pseudoinstructions

zext.w *rd*, *rs1* → add.uw *rd*, *rs1*, zero

Encoding



Description

This instruction performs an XLEN-wide addition between *rs2* and the zero-extended least-significant word of *rs1*.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + index;
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.2. andn

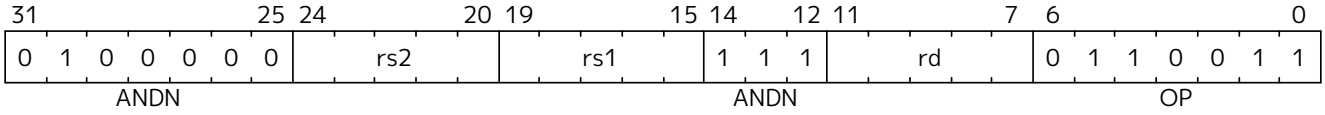
Synopsis

AND with inverted operand

Mnemonic

andn *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bitwise logical AND operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) & ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.3. bclr

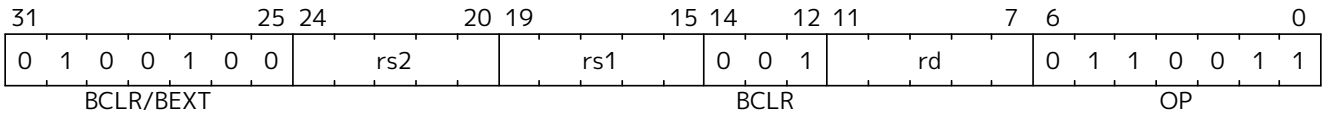
Synopsis

Single-Bit Clear (Register)

Mnemonic

bclr *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns *rs1* with a single bit cleared at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.4. bclri

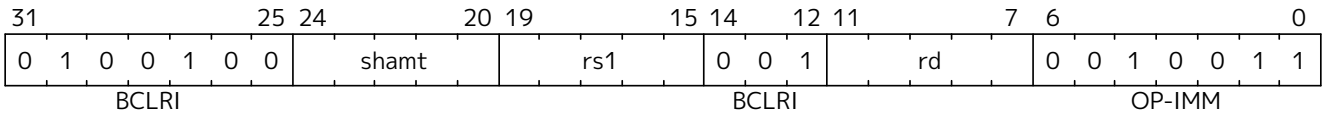
Synopsis

Single-Bit Clear (Immediate)

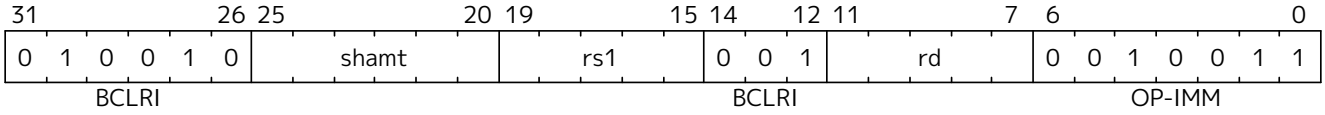
Mnemonic

bclri *rd*, *rs1*, *shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns *rs1* with a single bit cleared at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) & ~(1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.5. bext

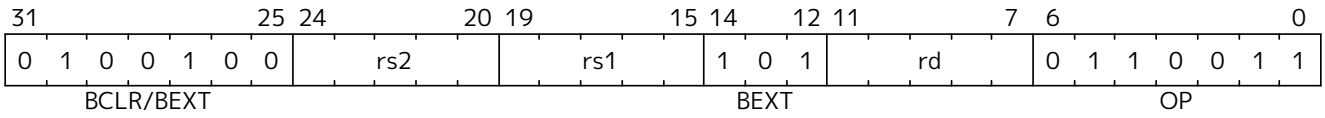
Synopsis

Single-Bit Extract (Register)

Mnemonic

bext *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns a single bit extracted from *rs1* at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.6. bexti

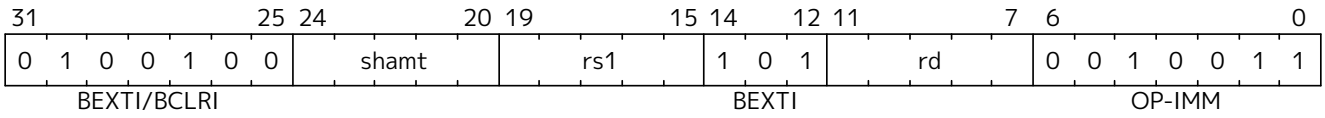
Synopsis

Single-Bit Extract (Immediate)

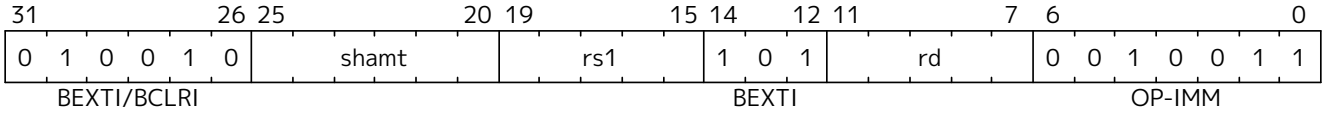
Mnemonic

bexti *rd*, *rs1*, *shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns a single bit extracted from *rs1* at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = (X(rs1) >> index) & 1;
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.7. binv

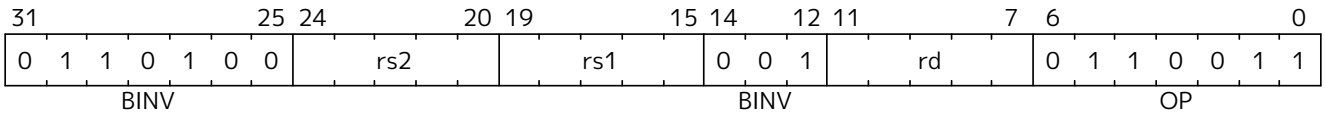
Synopsis

Single-Bit Invert (Register)

Mnemonic

binv *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns *rs1* with a single bit inverted at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.8. binvi

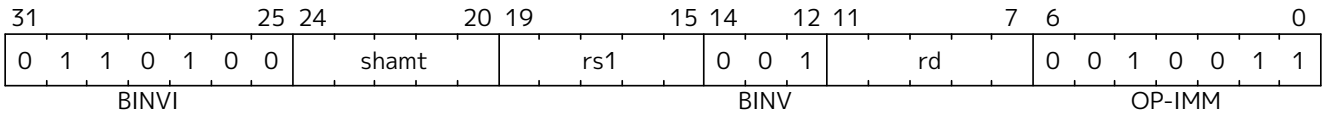
Synopsis

Single-Bit Invert (Immediate)

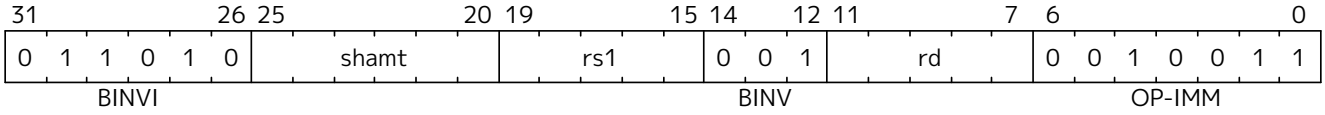
Mnemonic

binvi *rd*, *rs1*, *shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns *rs1* with a single bit inverted at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) ^ (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.9. bset

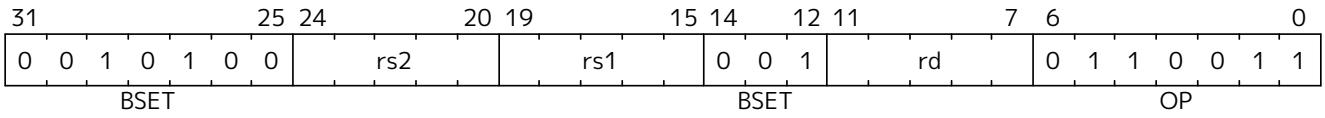
Synopsis

Single-Bit Set (Register)

Mnemonic

bset *rd*, *rs1*,*rs2*

Encoding



Description

This instruction returns *rs1* with a single bit set at the index specified in *rs2*. The index is read from the lower log2(XLEN) bits of *rs2*.

Operation

```
let index = X(rs2) & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.10. bseti

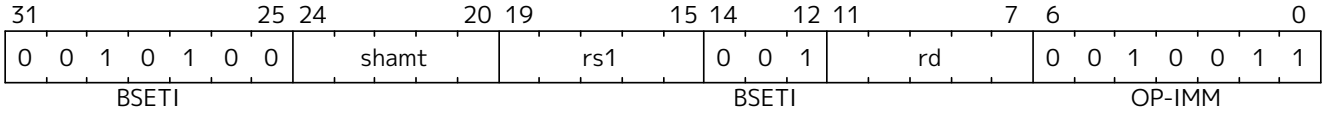
Synopsis

Single-Bit Set (Immediate)

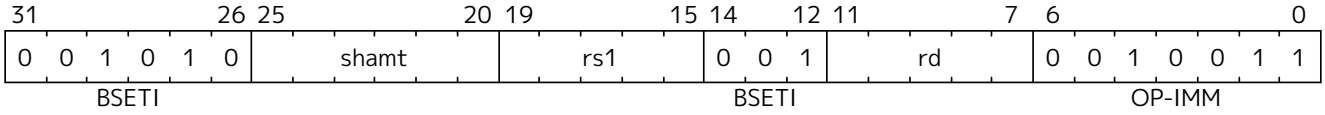
Mnemonic

bseti *rd*, *rs1*,*shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction returns *rs1* with a single bit set at the index specified in *shamt*. The index is read from the lower log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let index = shamt & (XLEN - 1);
X(rd) = X(rs1) | (1 << index)
```

Included in

Extension	Minimum version	Lifecycle state
Zbs (Single-bit instructions)	v1.0	Ratified

30.9.11. clmul

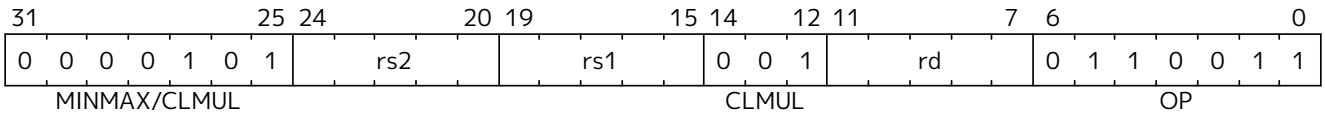
Synopsis

Carry-less multiply (low-part)

Mnemonic

clmul *rd*, *rs1*, *rs2*

Encoding



Description

clmul produces the lower half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val << i);
            else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	v1.0	Ratified
Zbkc (Carry-less multiplication for Cryptography)	v1.0	Ratified

30.9.12. clmulh

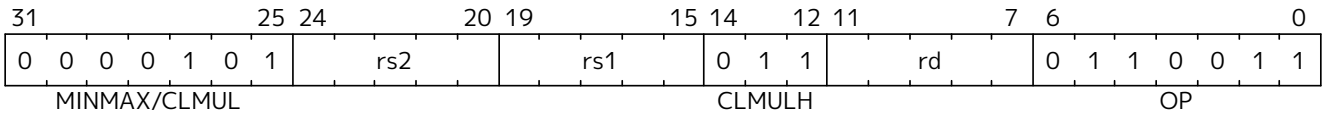
Synopsis

Carry-less multiply (high-part)

Mnemonic

clmulh *rd*, *rs1*, *rs2*

Encoding



Description

clmulh produces the upper half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 1 to xlen by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val >> (xlen - i));
            else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	v1.0	Ratified
Zbkc (Carry-less multiplication for Cryptography)	v1.0	Ratified

30.9.13. clmulr

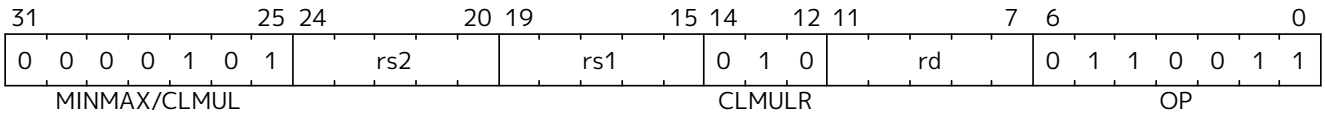
Synopsis

Carry-less multiply (reversed)

Mnemonic

clmulr *rd*, *rs1*, *rs2*

Encoding



Description

clmulr produces bits 2·XLEN−2:XLEN-1 of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val >> (xlen - i - 1));
            else output;
}

X[rd] = output
```

Note

The **clmulr** *instruction is used to accelerate CRC calculations. The* **r** *in the instruction's mnemonic stands for reversed, as the instruction is equivalent to bit-reversing the inputs, performing a* **clmul**, *then bit-reversing the output.*

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	v1.0	Ratified

30.9.14. clz

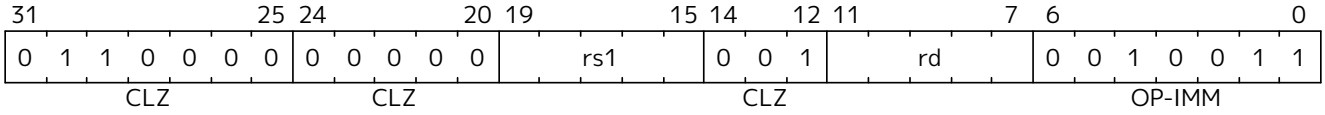
Synopsis

Count leading zero bits

Mnemonic

clz *rd*, *rs*

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN, and if the most-significant bit of the input is a 1, the output is 0.

Operation

```
val HighestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit x = {
  foreach (i from (xlen - 1) to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = (xlen - 1) - HighestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.15. clzw

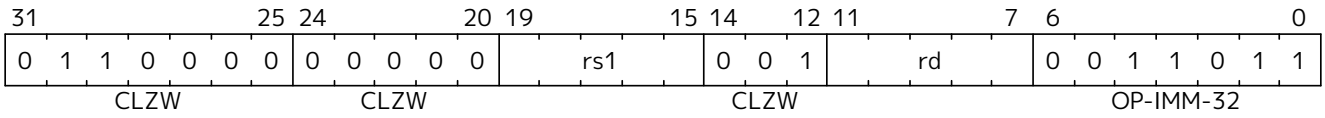
Synopsis

Count leading zero bits in word

Mnemonic

clzw *rd*, *rs*

Encoding



Description

This instruction counts the number of 0's before the first 1 starting at bit 31 and progressing to bit 0. Accordingly, if the least-significant word is 0, the output is 32, and if the most-significant bit of the word (i.e., bit 31) is a 1, the output is 0.

Operation

```
val HighestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function HighestSetBit32 x = {
  foreach (i from 31 to 0 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return -1;
}

let rs = X(rs);
X[rd] = 31 - HighestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.16. cpop

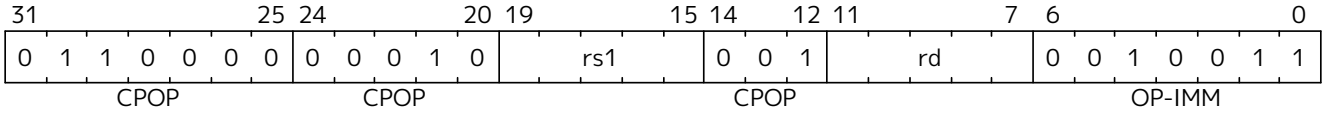
Synopsis

Count set bits

Mnemonic

cpop *rd*, *rs*

Encoding



Description

This instructions counts the number of 1's (i.e., set bits) in the source register.

Operation

```
let bitcount = 0;
let rs = X(rs);

foreach (i from 0 to (xlen - 1) in inc)
    if rs[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

Software Hint

This operation is known as population count, popcount, sideways sum, bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` *is implemented by cpop on RV32 and by* **cpopw** *on RV64. The GCC builtin function* `__builtin_popcountl (unsigned long x)` *for LP64 is implemented by* **cpop** *on RV64.*

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.17. cpopw

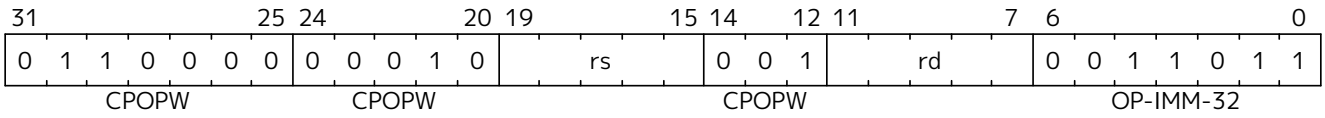
Synopsis

Count set bits in word

Mnemonic

cpopw *rd*, *rs*

Encoding



Description

This instructions counts the number of 1's (i.e., set bits) in the least-significant word of the source register.

Operation

```
let bitcount = 0;
let val = X(rs);

foreach (i from 0 to 31 in inc)
    if val[i] == 0b1 then bitcount = bitcount + 1 else ();

X[rd] = bitcount
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.18. ctz

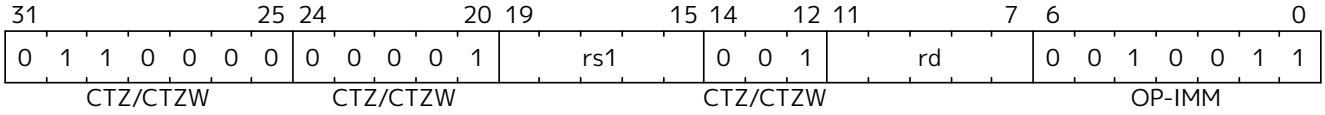
Synopsis

Count trailing zeros

Mnemonic

ctz *rd*, *rs*

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN, and if the least-significant bit of the input is a 1, the output is 0.

Operation

```
val LowestSetBit : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit x = {
  foreach (i from 0 to (xlen - 1) by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return xlen;
}

let rs = X(rs);
X[rd] = LowestSetBit(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.19. ctzw

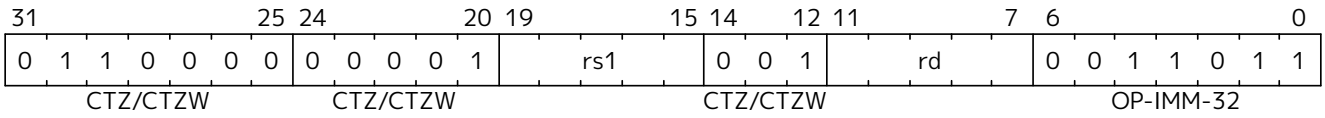
Synopsis

Count trailing zero bits in word

Mnemonic

ctzw *rd*, *rs*

Encoding



Description

This instruction counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit of the least-significant word (i.e., 31). Accordingly, if the least-significant word is 0, the output is 32, and if the least-significant bit of the input is a 1, the output is 0.

Operation

```
val LowestSetBit32 : forall ('N : Int), 'N >= 0. bits('N) -> int

function LowestSetBit32 x = {
  foreach (i from 0 to 31 by 1 in dec)
    if [x[i]] == 0b1 then return(i) else ();
  return 32;
}

let rs = X(rs);
X[rd] = LowestSetBit32(rs);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.20. max

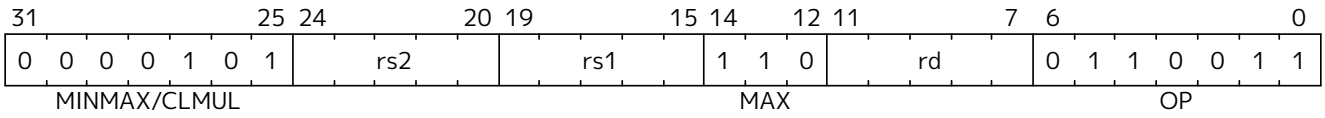
Synopsis

Maximum

Mnemonic

max *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns the larger of two signed integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs2_val
             else rs1_val;

X(rd) = result;
```

Software Hint

Calculating the absolute value of a signed integer can be performed using the following sequence: **neg rD,rS** *followed by* **max rD,rS,rD**. *When using this common sequence, it is suggested that they are scheduled with no intervening instructions so that implementations that are so optimized can fuse them together.*

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.21. maxu

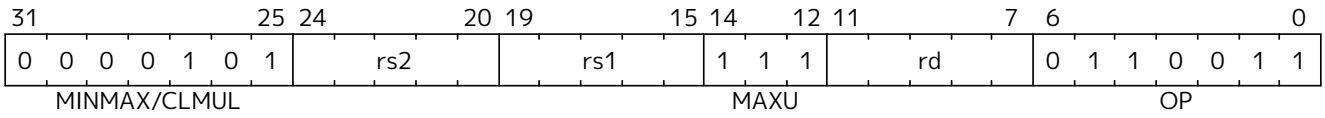
Synopsis

Unsigned maximum

Mnemonic

maxu *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns the larger of two unsigned integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs2_val
             else rs1_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.22. min

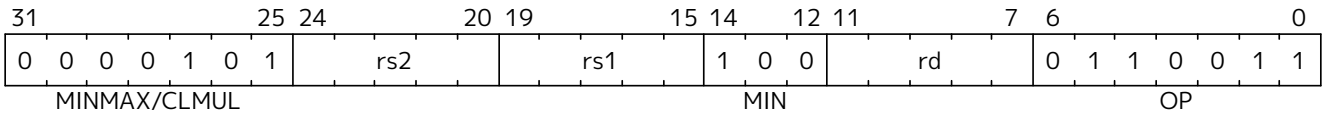
Synopsis

Minimum

Mnemonic

min *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns the smaller of two signed integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_s rs2_val
             then rs1_val
             else rs2_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.23. minu

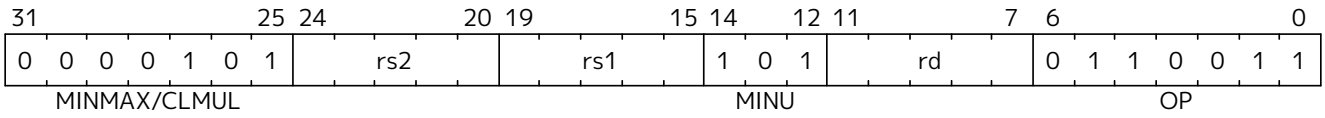
Synopsis

Unsigned minimum

Mnemonic

minu *rd*, *rs1*, *rs2*

Encoding



Description

This instruction returns the smaller of two unsigned integers.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);

let result = if   rs1_val <_u rs2_val
             then rs1_val
             else rs2_val;

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.24. orc.b

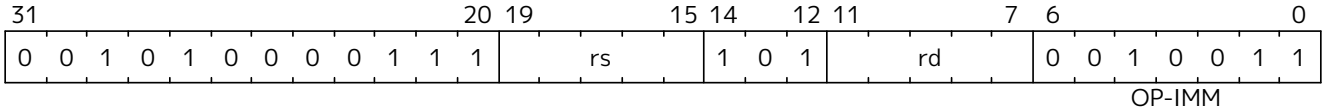
Synopsis

Bitwise OR-Combine, byte granule

Mnemonic

orc.b *rd*, *rs*

Encoding



Description

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result *rd* to all zeros if no bit within the respective byte of *rs* is set, or to all ones if any bit within the respective byte of *rs* is set.

Operation

```
let input = X(rs);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 8) by 8) {
   output[(i + 7)..i] = if   input[(i + 7)..i] == 0
                        then 0b00000000
                        else 0b11111111;
}

X[rd] = output;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified

30.9.25. orn

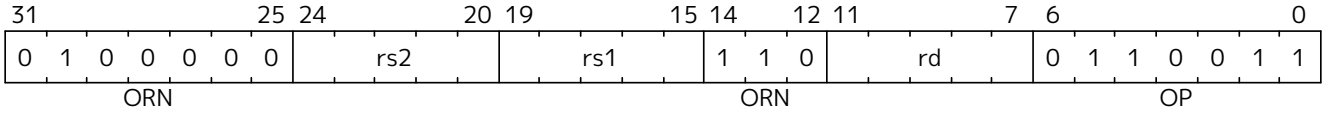
Synopsis

OR with inverted operand

Mnemonic

orn *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bitwise logical OR operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) | ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.26. pack

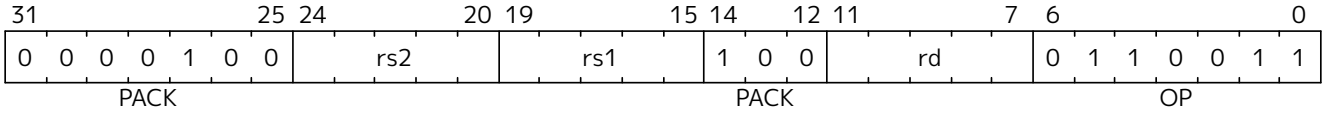
Synopsis

Pack the low halves of *rs1* and *rs2* into *rd*.

Mnemonic

pack *rd*, *rs1*, *rs2*

Encoding



Description

The pack instruction packs the XLEN/2-bit lower halves of *rs1* and *rs2* into *rd*, with *rs1* in the lower half and *rs2* in the upper half.

Operation

```
let lo_half : bits(xlen/2) = X(rs1)[xlen/2-1..0];
let hi_half : bits(xlen/2) = X(rs2)[xlen/2-1..0];
X(rd) = EXTZ(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

For RV32, the `pack` *instruction with rs2=*`x0` *is the* `zext.h` *instruction. Hence, for RV32, any extension that contains the* `pack` *instruction also contains the* `zext.h` *instruction (but not necessarily the* `c.zext.h` *instruction, which is only guaranteed to exist if both the Zcb and Zbb extensions are implemented).*

30.9.27. packh

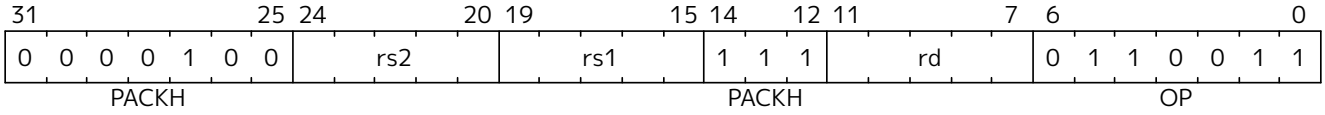
Synopsis

Pack the low bytes of *rs1* and *rs2* into *rd*.

Mnemonic

packh *rd*, *rs1*, *rs2*

Encoding



Description

The packh instruction packs the least-significant bytes of *rs1* and *rs2* into the 16 least-significant bits of *rd*, zero extending the rest of *rd*.

Operation

```
let lo_half : bits(8) = X(rs1)[7..0];
let hi_half : bits(8) = X(rs2)[7..0];
X(rd) = EXTZ(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.28. packw

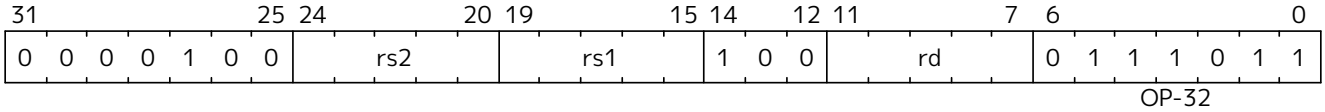
Synopsis

Pack the low 16-bits of *rs1* and *rs2* into *rd* on RV64.

Mnemonic

packw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction packs the low 16 bits of *rs1* and *rs2* into the 32 least-significant bits of *rd*, sign extending the 32-bit result to the rest of *rd*. This instruction only exists on RV64 based systems.

Operation

```
let lo_half : bits(16) = X(rs1)[15..0];
let hi_half : bits(16) = X(rs2)[15..0];
X(rd) = EXTS(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

For RV64, the `packw` *instruction with rs2=*`x0` *is the* `zext.h` *instruction. Hence, for RV64, any extension that contains the* `packw` *instruction also contains the* `zext.h` *instruction (but not necessarily the* `c.zext.h` *instruction, which is only guaranteed to exist if both the Zcb and Zbb extensions are implemented).*

30.9.29. rev8

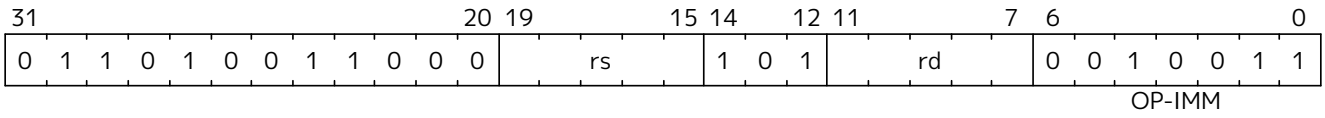
Synopsis

Byte-reverse register

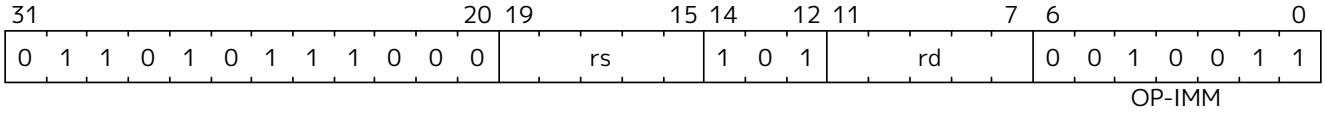
Mnemonic

rev8 *rd*, *rs*

Encoding (RV32)



Encoding (RV64)



Description

This instruction reverses the order of the bytes in *rs*.

Operation

```
let input = X(rs);
let output : xlenbits = 0;
let j = xlen - 1;

foreach (i from 0 to (xlen - 8) by 8) {
   output[i..(i + 7)] = input[(j - 7)..j];
   j = j - 8;
}

X[rd] = output
```

Note

The **rev8** *mnemonic corresponds to different instruction encodings in RV32 and RV64.*

Software Hint

The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 rd,rs` *followed by a* `srai rd,rd,K`*, where K is XLEN-32 and XLEN-16, respectively.*

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.30. brev8

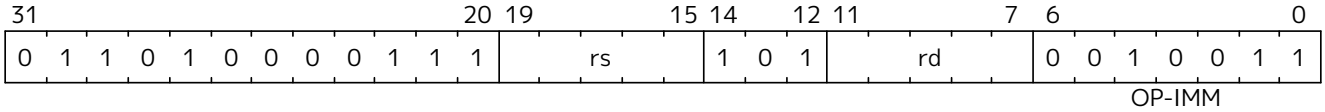
Synopsis

Reverse the bits in each byte of a source register.

Mnemonic

brev8 *rd*, *rs*

Encoding



Description

This instruction reverses the order of the bits in every byte of a register.

Operation

```
result : xlenbits = EXTZ(0b0);
foreach (i from 0 to sizeof(xlen) by 8) {
    result[i+7..i] = reverse_bits_in_byte(X(rs1)[i+7..i]);
};
X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.31. rol

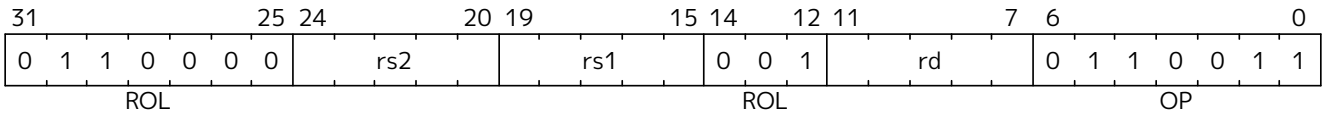
Synopsis

Rotate Left (Register)

Mnemonic

rol *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
            else X(rs2)[5..0];
let result = (X(rs1) << shamt) | (X(rs1) >> (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.32. rolw

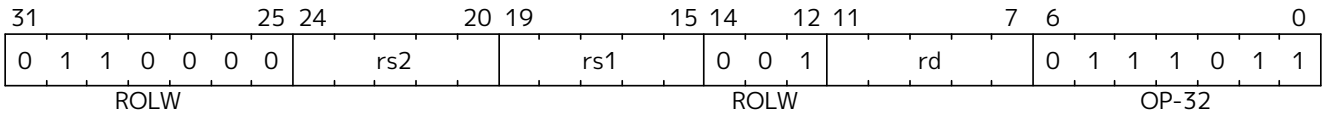
Synopsis

Rotate Left Word (Register)

Mnemonic

rolw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 << shamt) | (rs1 >> (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.33. ror

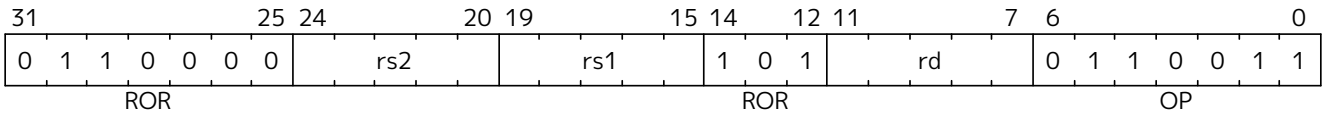
Synopsis

Rotate Right

Mnemonic

ror *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
            else X(rs2)[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.34. rori

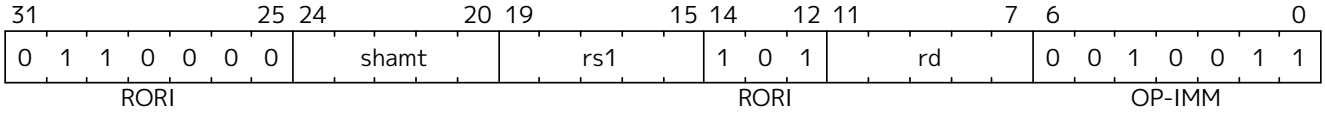
Synopsis

Rotate Right (Immediate)

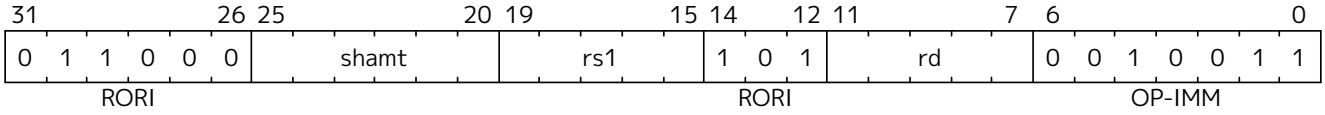
Mnemonic

rori *rd*, *rs1*, *shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction performs a rotate right of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let shamt = if   xlen == 32
            then shamt[4..0]
            else shamt[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.35. roriw

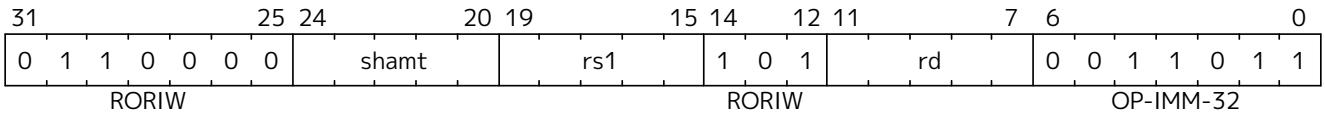
Synopsis

Rotate Right Word by Immediate

Mnemonic

roriw *rd*, *rs1*, *shamt*

Encoding



Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1_data = EXTZ(X(rs1)[31..0];
let result = (rs1_data >> shamt) | (rs1_data << (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.36. rorw

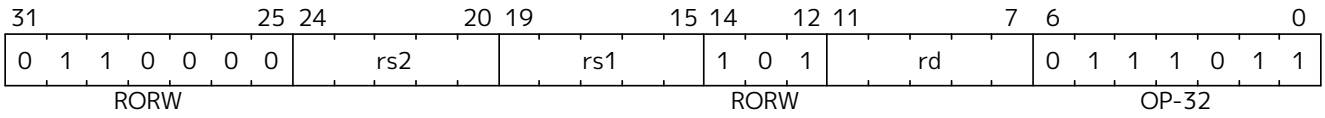
Synopsis

Rotate Right Word (Register)

Mnemonic

rorw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 >> shamt) | (rs1 << (32 - shamt));
X(rd) = EXTS(result);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.37. sext.b

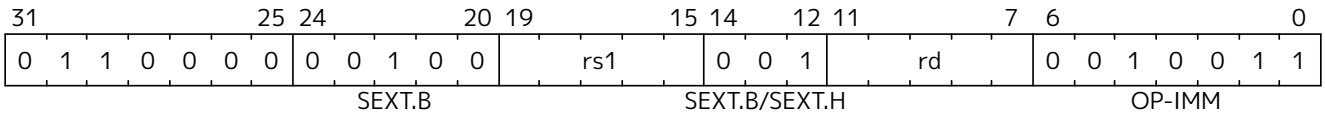
Synopsis

Sign-extend byte

Mnemonic

sext.b *rd*, *rs*

Encoding



Description

This instruction sign-extends the least-significant byte in the source to XLEN by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

Operation

```
X(rd) = EXTS(X(rs)[7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified

30.9.38. sext.h

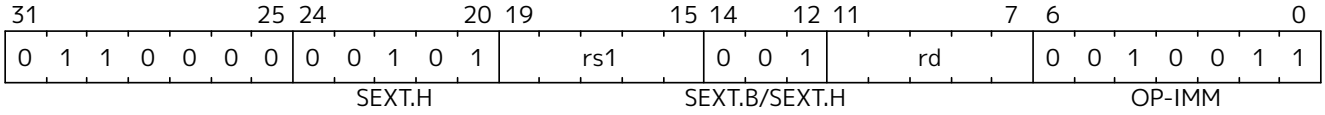
Synopsis

Sign-extend halfword

Mnemonic

sext.h *rd*, *rs*

Encoding



Description

This instruction sign-extends the least-significant halfword in *rs* to XLEN by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

Operation

```
X(rd) = EXTS(X(rs)[15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified

30.9.39. sh1add

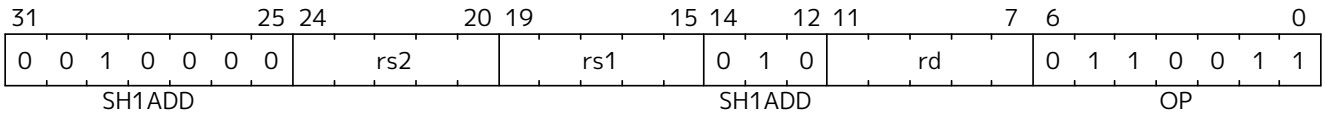
Synopsis

Shift left by 1 and add

Mnemonic

sh1add *rd*, *rs1*, *rs2*

Encoding



Description

This instruction shifts *rs1* to the left by 1 bit and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 1);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.40. sh1add.uw

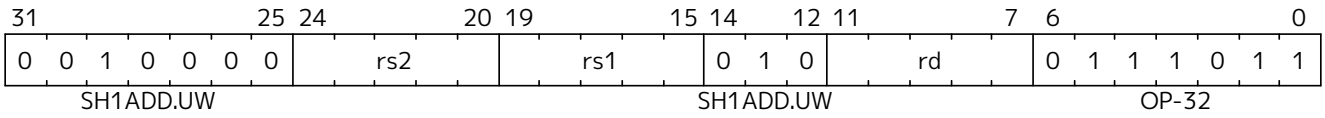
Synopsis

Shift unsigned word left by 1 and add

Mnemonic

sh1add.uw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 1 place.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 1);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.41. sh2add

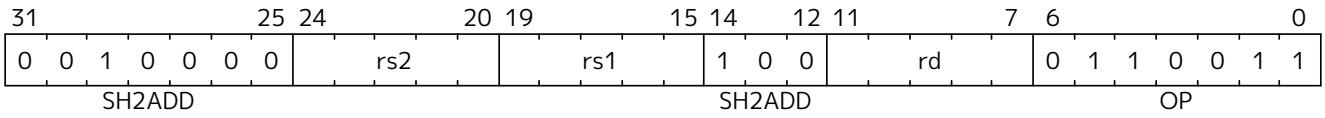
Synopsis

Shift left by 2 and add

Mnemonic

sh2add *rd*, *rs1*, *rs2*

Encoding



Description

This instruction shifts *rs1* to the left by 2 places and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 2);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.42. sh2add.uw

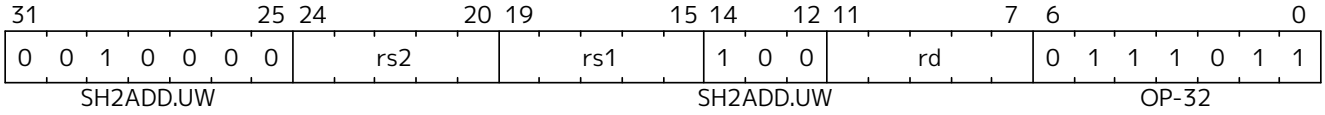
Synopsis

Shift unsigned word left by 2 and add

Mnemonic

sh2add.uw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 2 places.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 2);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.43. sh3add

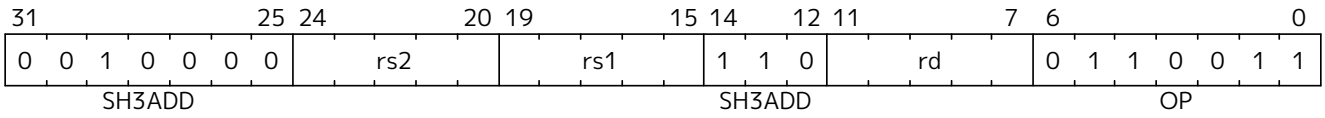
Synopsis

Shift left by 3 and add

Mnemonic

sh3add *rd*, *rs1*, *rs2*

Encoding



Description

This instruction shifts *rs1* to the left by 3 places and adds it to *rs2*.

Operation

```
X(rd) = X(rs2) + (X(rs1) << 3);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.44. sh3add.uw

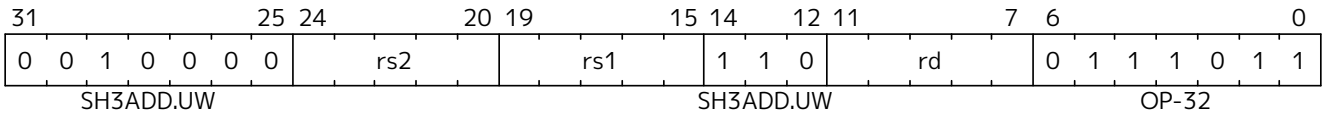
Synopsis

Shift unsigned word left by 3 and add

Mnemonic

sh3add.uw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs an XLEN-wide addition of two addends. The first addend is *rs2*. The second addend is the unsigned value formed by extracting the least-significant word of *rs1* and shifting it left by 3 places.

Operation

```
let base = X(rs2);
let index = EXTZ(X(rs1)[31..0]);

X(rd) = base + (index << 3);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

30.9.45. slli.uw

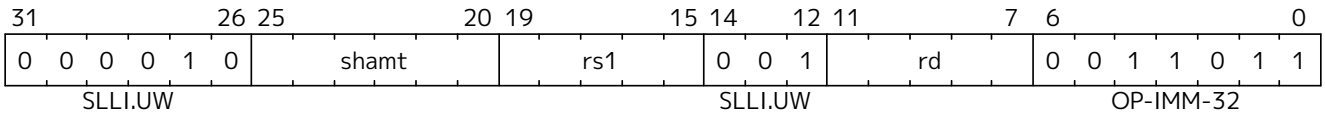
Synopsis

Shift-left unsigned word (Immediate)

Mnemonic

slli.uw *rd*, *rs1*, *shamt*

Encoding



Description

This instruction takes the least-significant word of *rs1*, zero-extends it, and shifts it left by the immediate.

Operation

```
X(rd) = (EXTZ(X(rs)[31..0]) << shamt);
```

Included in

Extension	Minimum version	Lifecycle state
Zba (Address generation instructions)	0.93	Ratified

Architecture Explanation

This instruction is the same as **slli** *with* **zext.w** *performed on rs1 before shifting.*

30.9.46. unzip

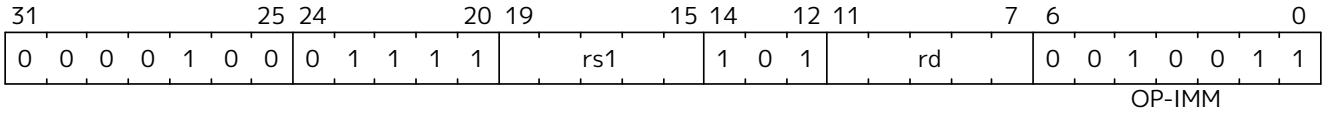
Synopsis

Place odd and even bits of the source register into upper and lower halves of the destination register, respectively.

Mnemonic

unzip *rd*, *rs*

Encoding



Description

This instruction scatters all of the odd and even bits of a source word into the high and low halves of a destination word. It is the inverse of the zip instruction. This instruction is available only on RV32.

Operation

```
foreach (i from 0 to xlen/2-1) {
  X(rd)[i] = X(rs1)[2*i]
  X(rd)[i+xlen/2] = X(rs1)[2*i+1]
}
```

Software Hint

This instruction is useful for implementing the SHA3 cryptographic hash function on a 32-bit architecture, as it implements the bit-interleaving operation used to speed up the 64-bit rotations directly.

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography) (RV32)	v1.0	Ratified

30.9.47. xnor

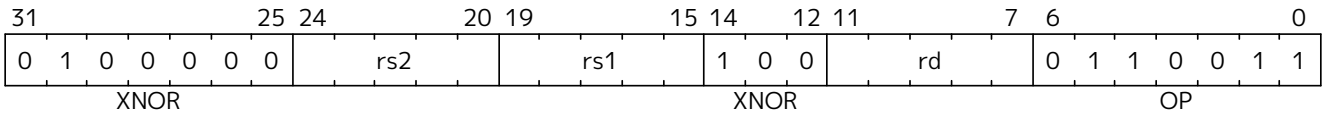
Synopsis

Exclusive NOR

Mnemonic

xnor *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bit-wise exclusive-NOR operation on *rs1* and *rs2*.

Operation

```
X(rd) = ~(X(rs1) ^ X(rs2));
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

30.9.48. xperm8

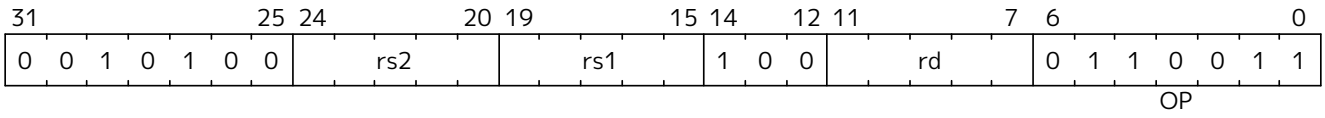
Synopsis

Byte-wise lookup of indices into a vector in registers.

Mnemonic

xperm8 *rd*, *rs1*, *rs2*

Encoding



Description

The xperm8 instruction operates on bytes. The *rs1* register contains a vector of XLEN/8 8-bit elements. The *rs2* register contains a vector of XLEN/8 8-bit indexes. The result is each element in *rs2* replaced by the indexed element in *rs1*, or zero if the index into *rs2* is out of bounds.

Operation

```
val xperm8_lookup : (bits(8), xlenbits) -> bits(8)
function xperm8_lookup (idx, lut) = {
    (lut >> (idx @ 0b000))[7..0]
}

function clause execute ( XPERM8 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 8) {
        result[i+7..i] = xperm8_lookup(X(rs2)[i+7..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zbkx (Crossbar permutations)	v1.0	Ratified

30.9.49. xperm4

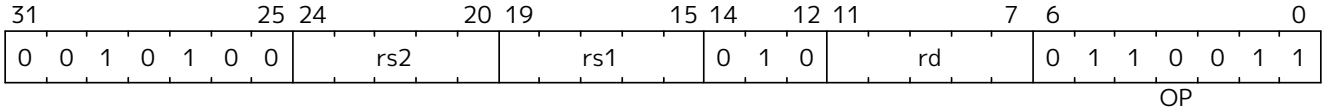
Synopsis

Nibble-wise lookup of indices into a vector.

Mnemonic

xperm4 *rd*, *rs1*, *rs2*

Encoding



Description

The xperm4 instruction operates on nibbles. The *rs1* register contains a vector of XLEN/4 4-bit elements. The *rs2* register contains a vector of XLEN/4 4-bit indexes. The result is each element in *rs2* replaced by the indexed element in *rs1*, or zero if the index into *rs2* is out of bounds.

Operation

```
val xperm4_lookup : (bits(4), xlenbits) -> bits(4)
function xperm4_lookup (idx, lut) = {
    (lut >> (idx @ 0b00))[3..0]
}

function clause execute ( XPERM4 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 4) {
        result[i+3..i] = xperm4_lookup(X(rs2)[i+3..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zbkx (Crossbar permutations)	v1.0	Ratified

30.9.50. zext.h

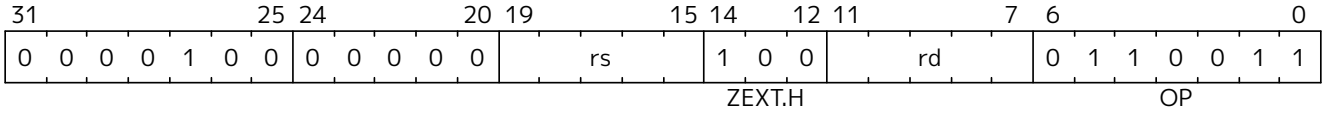
Synopsis

Zero-extend halfword

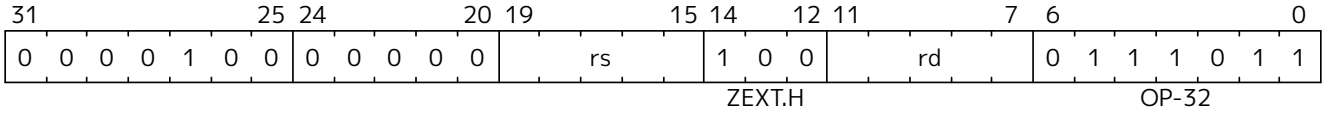
Mnemonic

zext.h *rd*, *rs*

Encoding (RV32)



Encoding (RV64)



Description

This instruction zero-extends the least-significant halfword of the source to XLEN by inserting 0's into all of the bits more significant than 15.

Operation

```
X(rd) = EXTZ(X(rs)[15..0]);
```

Note

The **zext.h** *mnemonic corresponds to different instruction encodings in RV32 and RV64.*

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified

30.9.51. zip

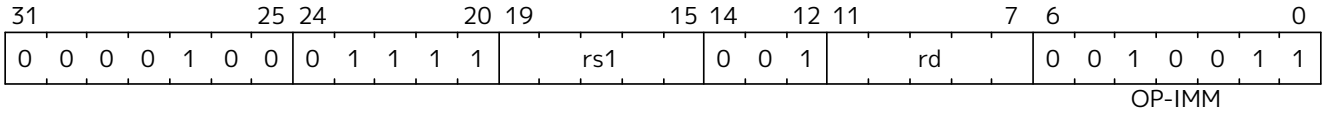
Synopsis

Interleave upper and lower halves of the source register into odd and even bits of the destination register, respectively.

Mnemonic

zip *rd*, *rs*

Encoding



Description

This instruction gathers bits from the high and low halves of the source word into odd/even bit positions in the destination word. It is the inverse of the unzip instruction. This instruction is available only on RV32.

Operation

```
foreach (i from 0 to xlen/2-1) {
  X(rd)[2*i] = X(rs1)[i]
  X(rd)[2*i+1] = X(rs1)[i+xlen/2]
}
```

Software Hint

This instruction is useful for implementing the SHA3 cryptographic hash function on a 32-bit architecture, as it implements the bit-interleaving operation used to speed up the 64-bit rotations directly.

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Bit-manipulation for Cryptography) (RV32)	v1.0	Ratified

Chapter 31. "V" Standard Extension for Vector Operations, Version 1.0

The base vector extension is intended to provide general support for data-parallel execution within the 32-bit instruction encoding space, with later vector extensions supporting richer functionality for certain domains.

31.1. Introduction

Section 31.18 lists the standard vector extensions and which instructions and element widths are supported by each extension.

31.2. Implementation-defined Constant Parameters

Each hart supporting a vector extension defines two parameters:

1. The maximum size in bits of a vector element that any operation can produce or consume, *ELEN* ≥ 8, which must be a power of 2.
2. The number of bits in a single vector register, *VLEN* ≥ ELEN, which must be a power of 2, and must be no greater than 2^{16}.

Standard vector extensions (Section 31.18) and architecture profiles may set further constraints on *ELEN* and *VLEN*.

Future extensions may allow ELEN > VLEN by holding one element using bits from multiple vector registers, but this extension does not include this option.

The upper limit on VLEN allows software to know that indices will fit into 16 bits (largest VLMAX of 65,536 occurs for LMUL=8 and SEW=8 with VLEN=65,536). Any future extension beyond 64Kib per vector register will require new configuration instructions such that software using the old configuration instructions does not see greater vector lengths.

The vector extension supports writing binary code that under certain constraints will execute portably on harts with different values for the VLEN parameter, provided the harts support the required element types and instructions.

Code can be written that will expose differences in implementation parameters.

In general, thread contexts with active vector state cannot be migrated during execution between harts that have any difference in VLEN or ELEN parameters.

31.3. Vector Extension Programmer's Model

The vector extension adds 32 vector registers, and seven unprivileged CSRs (`vstart`, `vxsat`, `vxrm`, `vcsr`, `vtype`, `vl`, `vlenb`) to a base scalar RISC-V ISA.

Table 42. New vector CSRs

Address	Privilege	Name	Description
0x008	URW	vstart	Vector start element index
0x009	URW	vxsat	Fixed-Point Saturate Flag

Address	Privilege	Name	Description
0x00A	URW	vxrm	Fixed-Point Rounding Mode
0x00F	URW	vcsr	Vector control and status register
0xC20	URO	vl	Vector length
0xC21	URO	vtype	Vector data type register
0xC22	URO	vlenb	VLEN/8 (vector register length in bytes)

The four CSR numbers `0x00B-0x00E` *are tentatively reserved for future vector CSRs, some of which may be mirrored into* `vcsr`.

31.3.1. Vector Registers

The vector extension adds 32 architectural vector registers, `v0-v31` to the base scalar RISC-V ISA.

Each vector register has a fixed VLEN bits of state.

31.3.2. Vector Context Status in `mstatus`

A vector context status field, `VS`, is added to `mstatus[10:9]` and shadowed in `sstatus[10:9]`. It is defined analogously to the floating-point context status field, `FS`.

Attempts to execute any vector instruction, or to access the vector CSRs, raise an illegal-instruction exception when `mstatus.VS` is set to Off.

When `mstatus.VS` is set to Initial or Clean, executing any instruction that changes vector state, including the vector CSRs, will change `mstatus.VS` to Dirty. Implementations may also change `mstatus.VS` from Initial or Clean to Dirty at any time, even when there is no change in vector state.

Accurate setting of `mstatus.VS` *is an optimization. Software will typically use VS to reduce context-swap overhead.*

If `mstatus.VS` is Dirty, `mstatus.SD` is 1; otherwise, `mstatus.SD` is set in accordance with existing specifications.

Implementations may have a writable `misa.V` field. Analogous to the way in which the floating-point unit is handled, the `mstatus.VS` field may exist even if `misa.V` is clear.

Allowing `mstatus.VS` *to exist when* `misa.V` *is clear, enables vector emulation and simplifies handling of* `mstatus.VS` *in systems with writable* `misa.V`.

31.3.3. Vector Context Status in `vsstatus`

When the hypervisor extension is present, a vector context status field, `VS`, is added to `vsstatus[10:9]`. It is defined analogously to the floating-point context status field, `FS`.

When V=1, both `vsstatus.VS` and `mstatus.VS` are in effect: attempts to execute any vector instruction, or to access the vector CSRs, raise an illegal-instruction exception when either field is set to Off.

When V=1 and neither `vsstatus.VS` nor `mstatus.VS` is set to Off, executing any instruction that changes vector state, including the vector CSRs, will change both `mstatus.VS` and `vsstatus.VS` to Dirty. Implementations may also change `mstatus.VS` or `vsstatus.VS` from Initial or Clean to Dirty at any time, even when there is no change in vector state.

If `vsstatus.VS` is Dirty, `vsstatus.SD` is 1; otherwise, `vsstatus.SD` is set in accordance with existing specifications.

If `mstatus.VS` is Dirty, `mstatus.SD` is 1; otherwise, `mstatus.SD` is set in accordance with existing specifications.

For implementations with a writable `misa.V` field, the `vsstatus.VS` field may exist even if `misa.V` is clear.

31.3.4. Vector Type (`vtype`) Register

The read-only XLEN-wide *vector type* CSR, `vtype` provides the default type used to interpret the contents of the vector register file, and can only be updated by `vset{i}vl{i}` instructions. The vector type determines the organization of elements in each vector register, and how multiple vector registers are grouped. The `vtype` register also indicates how masked-off elements and elements past the current vector length in a vector result are handled.

Allowing updates only via the `vset{i}vl{i}` instructions simplifies maintenance of the `vtype` register state.

The `vtype` register has five fields, `vill`, `vma`, `vta`, `vsew[2:0]`, and `vlmul[2:0]`. Bits `vtype[XLEN-2:8]` should be written with zero, and non-zero values in this field are reserved.

This diagram shows the layout for RV32 systems, whereas in general `vill` should be at bit XLEN-1.

Table 43. `vtype` register layout

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5:3	vsew[2:0]	Selected element width (SEW) setting
2:0	vlmul[2:0]	Vector register group multiplier (LMUL) setting

A small implementation supporting ELEN=32 requires only seven bits of state in `vtype`: two bits for `ma` and `ta`, two bits for `vsew[1:0]` and three bits for `vlmul[2:0]`. The illegal value represented by `vill` can be internally encoded using the illegal 64-bit combination in `vsew[1:0]` without requiring an additional storage bit to hold `vill`.

Further standard and custom vector extensions may extend these fields to support a greater variety of data types.

The primary motivation for the `vtype` CSR is to allow the vector instruction set to fit into a 32-bit instruction encoding space. A separate `vset{i}vl{i}` instruction can be used to set `vl` and/or `vtype` fields before execution of a vector instruction, and implementations may choose to fuse these two instructions into a single internal vector microop. In many cases, the `vl` and `vtype` values can be reused across multiple instructions, reducing the static and dynamic instruction overhead from the `vset{i}vl{i}` instructions. It is anticipated that a future extended 64-bit instruction encoding would allow these fields to be specified statically in the

instruction encoding.

31.3.4.1. Vector Selected Element Width (`vsew[2:0]`)

The value in `vsew` sets the dynamic *selected element width* (SEW). By default, a vector register is viewed as being divided into VLEN/SEW elements.

Table 44. vsew[2:0] (selected element width) encoding

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	X	X	Reserved

While it is anticipated the larger `vsew[2:0]` encodings (`100`-`111`) will be used to encode larger SEW, the encodings are formally reserved at this point.

Table 45. Example VLEN = 128 bits

SEW	Elements per vector register
64	2
32	4
16	8
8	16

The supported element width may vary with LMUL.

The current set of standard vector extensions do not vary supported element width with LMUL. Some future extensions may support larger SEWs only when bits from multiple vector registers are combined using LMUL. In this case, software that relies on large SEW should attempt to use the largest LMUL, and hence the fewest vector register groups, to increase the number of implementations on which the code will run. The `vill` bit in `vtype` should be checked after setting `vtype` to see if the configuration is supported, and an alternate code path should be provided if it is not. Alternatively, a profile can mandate the minimum SEW at each LMUL setting.

31.3.4.2. Vector Register Grouping (`vlmul[2:0]`)

Multiple vector registers can be grouped together, so that a single vector instruction can operate on multiple vector registers. The term *vector register group* is used herein to refer to one or more vector registers used as a single operand to a vector instruction. Vector register groups can be used to provide greater execution efficiency for longer application vectors, but the main reason for their inclusion is to allow double-width or larger elements to be operated on with the same vector length as single-width elements. The vector length multiplier, *LMUL*, when greater than 1, represents the default number of vector registers that are combined to form a vector register group. Implementations must support LMUL integer values of 1, 2, 4, and 8.

The vector architecture includes instructions that take multiple source and destination vector operands with different element widths, but the same number of elements. The effective LMUL (EMUL) of each vector operand is determined by the number of registers required to hold the

elements. For example, for a widening add operation, such as add 32-bit values to produce 64-bit results, a double-width result requires twice the LMUL of the single-width inputs.

LMUL can also be a fractional value, reducing the number of bits used in a single vector register. Fractional LMUL is used to increase the number of effective usable vector register groups when operating on mixed-width values.

With only integer LMUL values, a loop operating on a range of sizes would have to allocate at least one whole vector register (LMUL=1) for the narrowest data type and then would consume multiple vector registers (LMUL>1) to form a vector register group for each wider vector operand. This can limit the number of vector register groups available. With fractional LMUL, the widest values need occupy only a single vector register while narrower values can occupy a fraction of a single vector register, allowing all 32 architectural vector register names to be used for different values in a vector loop even when handling mixed-width values. Fractional LMUL implies portions of vector registers are unused, but in some cases, having more shorter register-resident vectors improves efficiency relative to fewer longer register-resident vectors.

Implementations must provide fractional LMUL settings that allow the narrowest supported type to occupy a fraction of a vector register corresponding to the ratio of the narrowest supported type's width to that of the largest supported type's width. In general, the requirement is to support LMUL ≥ SEW_{MIN}/ELEN, where SEW_{MIN} is the narrowest supported SEW value and ELEN is the widest supported SEW value. In the standard extensions, SEW_{MIN}=8. For standard vector extensions with ELEN=32, fractional LMULs of 1/2 and 1/4 must be supported. For standard vector extensions with ELEN=64, fractional LMULs of 1/2, 1/4, and 1/8 must be supported.

When LMUL < SEW_{MIN}/ELEN, there is no guarantee an implementation would have enough bits in the fractional vector register to store at least one element, as VLEN=ELEN is a valid implementation choice. For example, with VLEN=ELEN=32, and SEW_{MIN}=8, an LMUL of 1/8 would only provide four bits of storage in a vector register.

For a given supported fractional LMUL setting, implementations must support SEW settings between SEW_{MIN} and LMUL * ELEN, inclusive.

The use of `vtype` encodings with LMUL < SEW_{MIN}/ELEN is *reserved*, but implementations can set `vill` if they do not support these configurations.

Requiring all implementations to set `vill` in this case would prohibit future use of this case in an extension, so to allow for a future definition of LMUL<SEW_{MIN}/ELEN behavior, we consider the use of this case to be reserved.

It is recommended that assemblers provide a warning (not an error) if a `vsetvli` instruction attempts to write an LMUL < SEW_{MIN}/ELEN.

LMUL is set by the signed `vlmul` field in `vtype` (i.e., LMUL = $2^{\texttt{vlmul[2:0]}}$).

The derived value VLMAX = LMUL*VLEN/SEW represents the maximum number of elements that can be operated on with a single vector instruction given the current SEW and LMUL settings as shown in the table below.

vlmul[2:0]			LMUL	#groups	VLMAX	Registers grouped with register *n*
1	0	0	-	-	-	reserved
1	0	1	1/8	32	VLEN/SEW/8	`v` *n* (single register in group)
1	1	0	1/4	32	VLEN/SEW/4	`v` *n* (single register in group)

vlmul[2:0]			LMUL	#groups	VLMAX	Registers grouped with register *n*
1	1	1	1/2	32	VLEN/SEW/2	**v** *n* (single register in group)
0	0	0	1	32	VLEN/SEW	**v** *n* (single register in group)
0	0	1	2	16	2*VLEN/SEW	**v** *n*, **v** *n*+1
0	1	0	4	8	4*VLEN/SEW	**v** *n*, ..., **v** *n*+3
0	1	1	8	4	8*VLEN/SEW	**v** *n*, ..., **v** *n*+7

When LMUL=2, the vector register group contains vector register **v** *n* and vector register **v** *n*+1, providing twice the vector length in bits. Instructions specifying an LMUL=2 vector register group with an odd-numbered vector register are reserved.

When LMUL=4, the vector register group contains four vector registers, and instructions specifying an LMUL=4 vector register group using vector register numbers that are not multiples of four are reserved.

When LMUL=8, the vector register group contains eight vector registers, and instructions specifying an LMUL=8 vector register group using register numbers that are not multiples of eight are reserved.

Mask registers are always contained in a single vector register, regardless of LMUL.

31.3.4.3. Vector Tail Agnostic and Vector Mask Agnostic `vta` and `vma`

These two bits modify the behavior of destination tail elements and destination inactive masked-off elements respectively during the execution of vector instructions. The tail and inactive sets contain element positions that are not receiving new results during a vector operation, as defined in Section 31.5.4.

All systems must support all four options:

`vta`	`vma`	Tail Elements	Inactive Elements
0	0	undisturbed	undisturbed
0	1	undisturbed	agnostic
1	0	agnostic	undisturbed
1	1	agnostic	agnostic

Mask destination tail elements are always treated as tail-agnostic, regardless of the setting of `vta`.

When a set is marked undisturbed, the corresponding set of destination elements in a vector register group retain the value they previously held.

When a set is marked agnostic, the corresponding set of destination elements in any vector destination operand can either retain the value they previously held, or are overwritten with 1s. Within a single vector instruction, each destination element can be either left undisturbed or overwritten with 1s, in any combination, and the pattern of undisturbed or overwritten with 1s is not required to be deterministic when the instruction is executed with the same inputs.

The agnostic policy was added to accommodate machines with vector register renaming. With an undisturbed policy, all elements would have to be read from the old physical destination vector register to be copied into the new physical destination vector register. This causes an inefficiency when these inactive or tail values are not required for subsequent calculations.

The value of all 1s instead of all 0s was chosen for the overwrite value to discourage software developers from depending on the value written.

A simple in-order implementation can ignore the settings and simply execute all vector instructions using the undisturbed policy. The `vta` *and* `vma` *state bits must still be provided in* `vtype` *for compatibility and to support thread migration.*

An out-of-order implementation can choose to implement tail-agnostic + mask-agnostic using tail-agnostic + mask-undisturbed to reduce implementation complexity.

The definition of agnostic result policy is left loose to accommodate migrating application threads between harts on a small in-order core (which probably leaves agnostic regions undisturbed) and harts on a larger out-of-order core with register renaming (which probably overwrites agnostic elements with 1s). As it might be necessary to restart in the middle, we allow arbitrary mixing of agnostic policies within a single vector instruction. This allowed mixing of policies also enables implementations that might change policies for different granules of a vector register, for example, using undisturbed within a granule that is actively operated on but renaming to all 1s for granules in the tail.

In addition, except for mask load instructions, any element in the tail of a mask result can also be written with the value the mask-producing operation would have calculated with `vl`=VLMAX. Furthermore, for mask-logical instructions and `vmsbf.m`, `vmsif.m`, `vmsof.m` mask-manipulation instructions, any element in the tail of the result can be written with the value the mask-producing operation would have calculated with `vl`=VLEN, SEW=8, and LMUL=8 (i.e., all bits of the mask register can be overwritten).

Mask tails are always treated as agnostic to reduce complexity of managing mask data, which can be written at bit granularity. There appears to be little software need to support tail-undisturbed for mask register values. Allowing mask-generating instructions to write back the result of the instruction avoids the need for logic to mask out the tail, except mask loads cannot write memory values to destination mask tails as this would imply accessing memory past software intent.

The assembly syntax adds two mandatory flags to the `vsetvli` instruction:

```
ta   # Tail agnostic
tu   # Tail undisturbed
ma   # Mask agnostic
mu   # Mask undisturbed

vsetvli t0, a0, e32, m4, ta, ma   # Tail agnostic, mask agnostic
vsetvli t0, a0, e32, m4, tu, ma   # Tail undisturbed, mask agnostic
vsetvli t0, a0, e32, m4, ta, mu   # Tail agnostic, mask undisturbed
vsetvli t0, a0, e32, m4, tu, mu   # Tail undisturbed, mask undisturbed
```

Prior to v0.9, when these flags were not specified on a `vsetvli`*, they defaulted to mask-undisturbed/tail-undisturbed. The use of* `vsetvli` *without these flags is deprecated, however, and specifying a flag setting is now mandatory. The default should perhaps be tail-agnostic/mask-agnostic, so software has to specify when it cares about the non-participating elements, but given the historical meaning of the instruction prior to introduction of these flags, it was decided to always require them in future assembly code.*

31.3.4.4. Vector Type Illegal (`vill`)

The `vill` bit is used to encode that a previous `vset{i}vl{i}` instruction attempted to write an unsupported value to `vtype`.

The `vill` bit is held in bit XLEN-1 of the CSR to support checking for illegal values with a branch on the sign bit.

If the `vill` bit is set, then any attempt to execute a vector instruction that depends upon `vtype` will raise an illegal-instruction exception.

`vset{i}vl{i}` *and whole register loads and stores do not depend upon* `vtype`.

When the `vill` bit is set, the other XLEN-1 bits in `vtype` shall be zero.

31.3.5. Vector Length (`vl`) Register

The *XLEN*-bit-wide read-only `vl` CSR can only be updated by the `vset{i}vl{i}` instructions, and the *fault-only-first* vector load instruction variants.

The `vl` register holds an unsigned integer specifying the number of elements to be updated with results from a vector instruction, as further detailed in Section 31.5.4.

The number of bits implemented in `vl` depends on the implementation's maximum vector length of the smallest supported type. The smallest vector implementation with VLEN=32 and supporting SEW=8 would need at least six bits in `vl` to hold the values 0-32 (VLEN=32, with LMUL=8 and SEW=8, yields VLMAX=32).

31.3.6. Vector Byte Length (`vlenb`) Register

The *XLEN*-bit-wide read-only CSR `vlenb` holds the value VLEN/8, i.e., the vector register length in bytes.

The value in `vlenb` is a design-time constant in any implementation.

Without this CSR, several instructions are needed to calculate VLEN in bytes, and the code has to disturb current `vl` and `vtype` settings which require them to be saved and restored.

31.3.7. Vector Start Index (`vstart`) Register

The *XLEN*-bit-wide read-write `vstart` CSR specifies the index of the first element to be executed by a vector instruction, as described in Section 31.5.4.

Normally, `vstart` is only written by hardware on a trap on a vector instruction, with the `vstart` value representing the element on which the trap was taken (either a synchronous exception or an asynchronous interrupt), and at which execution should resume after a resumable trap is handled.

All vector instructions are defined to begin execution with the element number given in the `vstart` CSR, leaving earlier elements in the destination vector undisturbed, and to reset the `vstart` CSR to zero at the end of execution.

All vector instructions, including `vset{i}vl{i}`, reset the `vstart` CSR to zero.

`vstart` is not modified by vector instructions that raise illegal-instruction exceptions.

The `vstart` CSR is defined to have only enough writable bits to hold the largest element index (one less than the maximum VLMAX).

*The maximum vector length is obtained with the largest LMUL setting (8) and the smallest SEW setting (8), so VLMAX_max = 8*VLEN/8 = VLEN. For example, for VLEN=256, `vstart`*

would have 8 bits to represent indices from 0 through 255.

The use of `vstart` values greater than the largest element index for the current `vtype` setting is reserved.

It is recommended that implementations trap if `vstart` is out of bounds. It is not required to trap, as a possible future use of upper `vstart` bits is to store imprecise trap information.

The `vstart` CSR is writable by unprivileged code, but non-zero `vstart` values may cause vector instructions to run substantially slower on some implementations, so `vstart` should not be used by application programmers. A few vector instructions cannot be executed with a non-zero `vstart` value and will raise an illegal-instruction exception as defined below.

Making `vstart` visible to unprivileged code supports user-level threading libraries.

Implementations are permitted to raise illegal-instruction exceptions when attempting to execute a vector instruction with a value of `vstart` that the implementation can never produce when executing that same instruction with the same `vtype` setting.

For example, some implementations will never take interrupts during execution of a vector arithmetic instruction, instead waiting until the instruction completes to take the interrupt. Such implementations are permitted to raise an illegal-instruction exception when attempting to execute a vector arithmetic instruction when `vstart` is nonzero.

When migrating a software thread between two harts with different microarchitectures, the `vstart` value might not be supported by the new hart microarchitecture. The runtime on the receiving hart might then have to emulate instruction execution up to the next supported `vstart` element position. Alternatively, migration events can be constrained to only occur at mutually supported `vstart` locations.

31.3.8. Vector Fixed-Point Rounding Mode (`vxrm`) Register

The vector fixed-point rounding-mode register holds a two-bit read-write rounding-mode field in the least-significant bits (`vxrm[1:0]`). The upper bits, `vxrm[XLEN-1:2]`, should be written as zeros.

The vector fixed-point rounding-mode is given a separate CSR address to allow independent access, but is also reflected as a field in `vcsr`.

A new rounding mode can be set while saving the original rounding mode using a single `csrwi` instruction.

The fixed-point rounding algorithm is specified as follows. Suppose the pre-rounding result is `v`, and `d` bits of that result are to be rounded off. Then the rounded result is `(v >> d) + r`, where `r` depends on the rounding mode as specified in the following table.

Table 46. vxrm encoding

`vxrm[1:0]`		Abbreviation	Rounding Mode	Rounding increment, `r`
0	0	rnu	round-to-nearest-up (add +0.5 LSB)	`v[d-1]`
0	1	rne	round-to-nearest-even	`v[d-1] & (v[d-2:0]≠0 | v[d])`
1	0	rdn	round-down	`0`
1	1	rod	round-to-odd (OR bits into LSB, aka "jam")	`!v[d] & v[d-1:0]≠0`

The rounding functions:

```
roundoff_unsigned(v, d) = (unsigned(v) >> d) + r
roundoff_signed(v, d) = (signed(v) >> d) + r
```

are used to represent this operation in the instruction descriptions below.

31.3.9. Vector Fixed-Point Saturation Flag (vxsat)

The `vxsat` CSR has a single read-write least-significant bit (`vxsat[0]`) that indicates if a fixed-point instruction has had to saturate an output value to fit into a destination format. Bits `vxsat[XLEN-1:1]` should be written as zeros.

The `vxsat` bit is mirrored in `vcsr`.

31.3.10. Vector Control and Status (vcsr) Register

The `vxrm` and `vxsat` separate CSRs can also be accessed via fields in the *XLEN*-bit-wide vector control and status CSR, `vcsr`.

Table 47. vcsr layout

Bits	Name	Description
XLEN-1:3		Reserved
2:1	vxrm[1:0]	Fixed-point rounding mode
0	vxsat	Fixed-point accrued saturation flag

31.3.11. State of Vector Extension at Reset

The vector extension must have a consistent state at reset. In particular, `vtype` and `vl` must have values that can be read and then restored with a single `vsetvl` instruction.

It is recommended that at reset, `vtype.vill` is set, the remaining bits in `vtype` are zero, and `vl` is set to zero.

The `vstart`, `vxrm`, `vxsat` CSRs can have arbitrary values at reset.

Most uses of the vector unit will require an initial `vset{i}vl{i}`, which will reset `vstart`. The `vxrm` and `vxsat` fields should be reset explicitly in software before use.

The vector registers can have arbitrary values at reset.

31.4. Mapping of Vector Elements to Vector Register State

The following diagrams illustrate how different width elements are packed into the bytes of a vector register depending on the current SEW and LMUL settings, as well as implementation VLEN. Elements are packed into each vector register with the least-significant byte in the lowest-numbered bits.

The mapping was chosen to provide the simplest and most portable model for software, but might appear to incur large wiring cost for wider vector datapaths on certain operations. The vector instruction set was expressly designed to support implementations that internally rearrange vector data for different SEW to reduce datapath wiring costs, while externally preserving the simple software model.

For example, microarchitectures can track the EEW with which a vector register was written, and then insert additional scrambling operations to rearrange data if the register is accessed with a different EEW.

31.4.1. Mapping for LMUL = 1

When LMUL=1, elements are simply packed in order from the least-significant to most-significant bits of the vector register.

To increase readability, vector register layouts are drawn with bytes ordered from right to left with increasing byte address. Bits within an element are numbered in a little-endian format with increasing bit index from right to left corresponding to increasing magnitude.

```
LMUL=1 examples.

The element index is given in hexadecimal and is shown placed at the
least-significant byte of the stored element.


 VLEN=32b

 Byte         3 2 1 0

 SEW=8b       3 2 1 0
 SEW=16b        1   0
 SEW=32b            0

 VLEN=64b

 Byte        7 6 5 4 3 2 1 0

 SEW=8b      7 6 5 4 3 2 1 0
 SEW=16b       3   2   1   0
 SEW=32b           1       0
 SEW=64b                   0

 VLEN=128b

 Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0

 SEW=8b      F E D C B A 9 8 7 6 5 4 3 2 1 0
 SEW=16b       7   6   5   4   3   2   1   0
 SEW=32b           3       2       1       0
 SEW=64b                   1               0

 VLEN=256b

 Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0

 SEW=8b   1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
```

```
SEW=16b     F   E   D   C   B   A   9   8   7   6   5   4   3   2   1   0
SEW=32b         7       6       5       4       3       2       1       0
SEW=64b                 3               2               1               0
```

31.4.2. Mapping for LMUL < 1

When LMUL < 1, only the first LMUL*VLEN/SEW elements in the vector register are used. The remaining space in the vector register is treated as part of the tail, and hence must obey the vta setting.

```
Example, VLEN=128b, LMUL=1/4

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0

SEW=8b      - - - - - - - - - - - - 3 2 1 0
SEW=16b       -   -   -   -   -   -   1   0
SEW=32b           -       -       -       0
```

31.4.3. Mapping for LMUL > 1

When vector registers are grouped, the elements of the vector register group are packed contiguously in element order beginning with the lowest-numbered vector register and moving to the next-highest-numbered vector register in the group once each vector register is filled.

```
LMUL > 1 examples

VLEN=32b, SEW=8b, LMUL=2

Byte         3 2 1 0
v2*n         3 2 1 0
v2*n+1       7 6 5 4

VLEN=32b, SEW=16b, LMUL=2

Byte         3 2 1 0
v2*n           1   0
v2*n+1         3   2

VLEN=32b, SEW=16b, LMUL=4

Byte         3 2 1 0
v4*n           1   0
v4*n+1         3   2
v4*n+2         5   4
v4*n+3         7   6

VLEN=32b, SEW=32b, LMUL=4
```

```
Byte         3 2 1 0
v4*n               0
v4*n+1             1
v4*n+2             2
v4*n+3             3

VLEN=64b, SEW=32b, LMUL=2

Byte         7 6 5 4 3 2 1 0
v2*n               1       0
v2*n+1             3       2

VLEN=64b, SEW=32b, LMUL=4

Byte         7 6 5 4 3 2 1 0
v4*n               1       0
v4*n+1             3       2
v4*n+2             5       4
v4*n+3             7       6

VLEN=128b, SEW=32b, LMUL=2

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n              3       2       1       0
v2*n+1            7       6       5       4

VLEN=128b, SEW=32b, LMUL=4

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                3       2       1       0
v4*n+1              7       6       5       4
v4*n+2              B       A       9       8
v4*n+3              F       E       D       C
```

31.4.4. Mapping across Mixed-Width Operations

The vector ISA is designed to support mixed-width operations without requiring additional explicit rearrangement instructions. The recommended software strategy when operating on multiple vectors with different precision values is to modify `vtype` dynamically to keep SEW/LMUL constant (and hence VLMAX constant).

The following example shows four different packed element widths (8b, 16b, 32b, 64b) in a VLEN=128b implementation. The vector register grouping factor (LMUL) is increased by the relative element size such that each group can hold the same number of vector elements (VLMAX=8 in this example) to simplify strip-mining code.

```
Example VLEN=128b, with SEW/LMUL=16

Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0
```

```
vn        - - - - - - - - 7 6 5 4 3 2 1 0  SEW=8b, LMUL=1/2

vn          7   6   5   4   3   2   1   0  SEW=16b, LMUL=1

v2*n            3       2       1       0  SEW=32b, LMUL=2
v2*n+1          7       6       5       4

v4*n                    1               0  SEW=64b, LMUL=4
v4*n+1                  3               2
v4*n+2                  5               4
v4*n+3                  7               6
```

The following table shows each possible constant SEW/LMUL operating point for loops with mixed-width operations. Each column represents a constant SEW/LMUL operating point. Entries in table are the LMUL values that yield that column's SEW/LMUL value for the data width on that row. In each column, an LMUL setting for a data width indicates that it can be aligned with the other data widths in the same column that also have an LMUL setting, such that all have the same VLMAX.

	SEW/LMUL						
	1	2	4	8	16	32	64
SEW= 8	8	4	2	1	1/2	1/4	1/8
SEW= 16		8	4	2	1	1/2	1/4
SEW= 32			8	4	2	1	1/2
SEW= 64				8	4	2	1

Larger LMUL settings can also used to simply increase vector length to reduce instruction fetch and dispatch overheads in cases where fewer vector register groups are needed.

31.4.5. Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL.

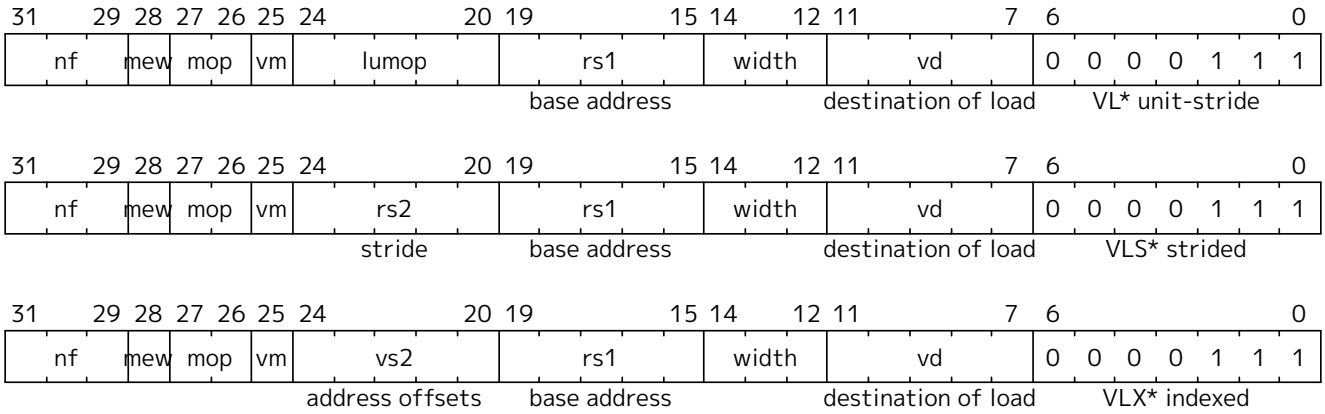
Each element is allocated a single mask bit in a mask vector register. The mask bit for element *i* is located in bit *i* of the mask register, independent of SEW or LMUL.

31.5. Vector Instruction Formats

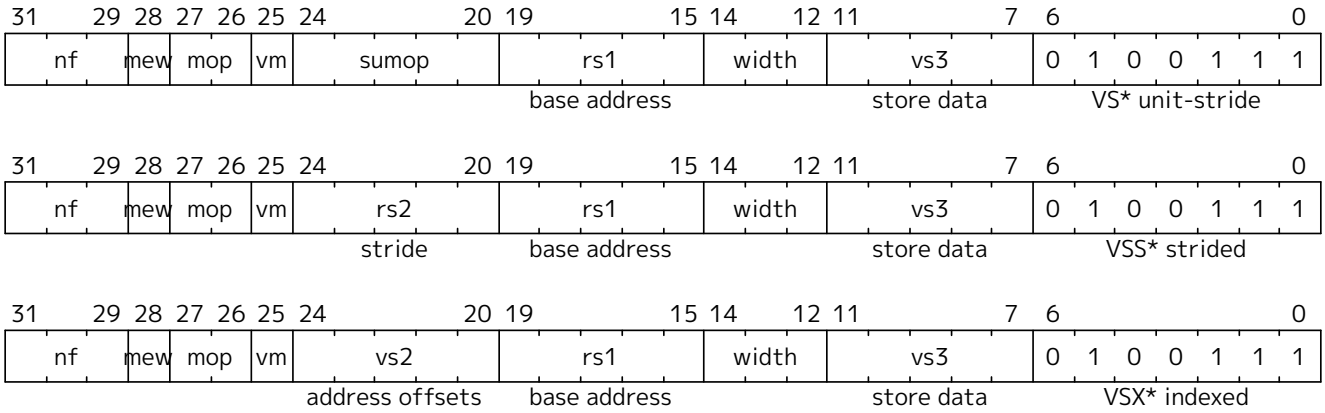
The instructions in the vector extension fit under two existing major opcodes (LOAD-FP and STORE-FP) and one new major opcode (OP-V).

Vector loads and stores are encoded within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP). The vector load and store encodings repurpose a portion of the standard scalar floating-point load/store 12-bit immediate field to provide further vector instruction encoding, with bit 25 holding the standard vector mask bit (see Section 31.5.3.1).

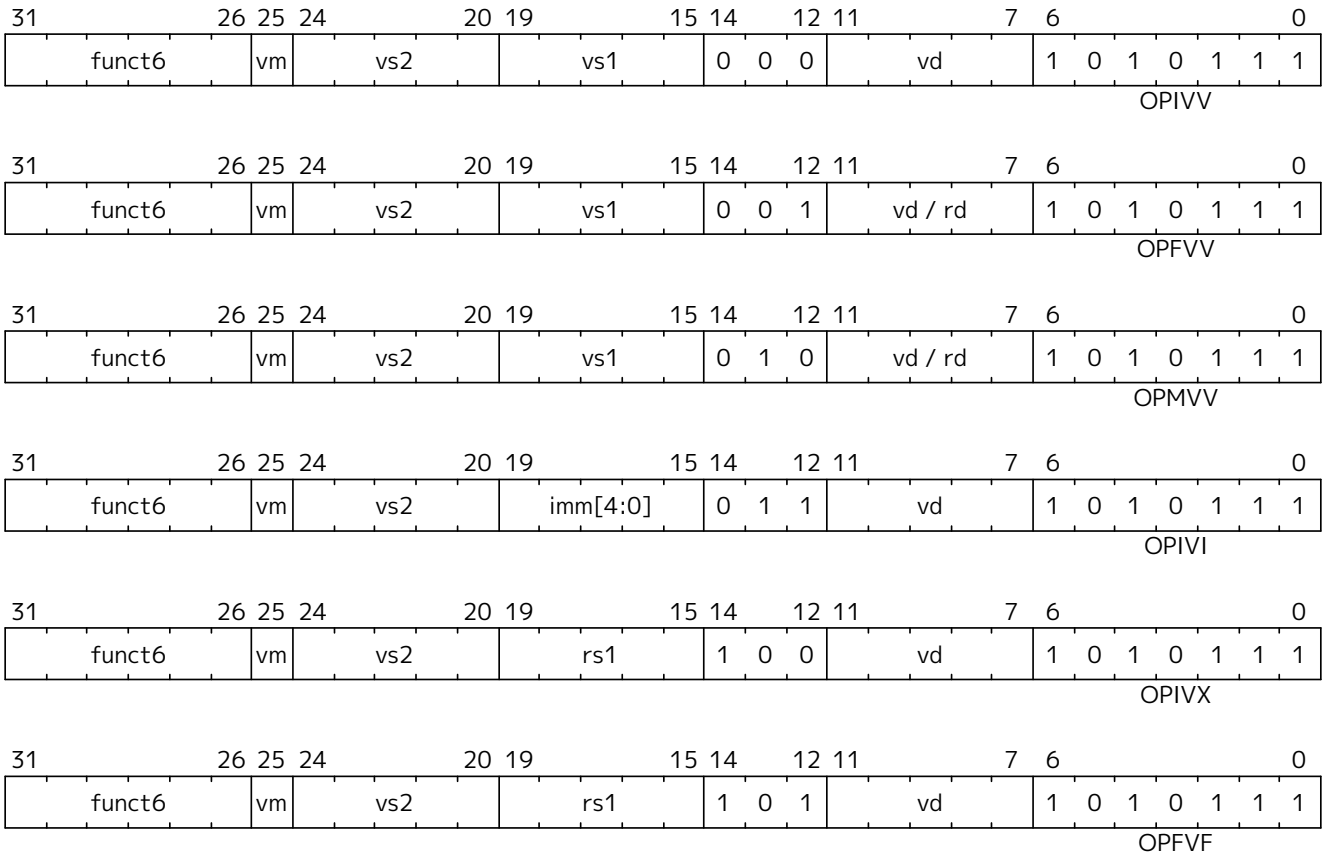
Format for Vector Load Instructions under LOAD-FP major opcode

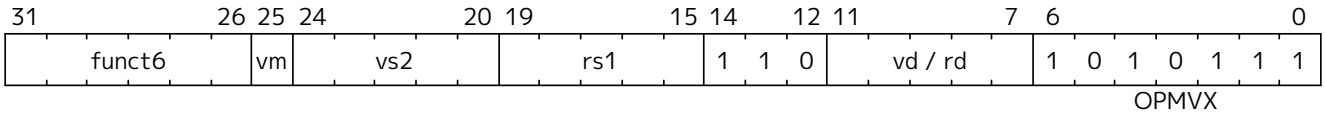


Format for Vector Store Instructions under STORE-FP major opcode

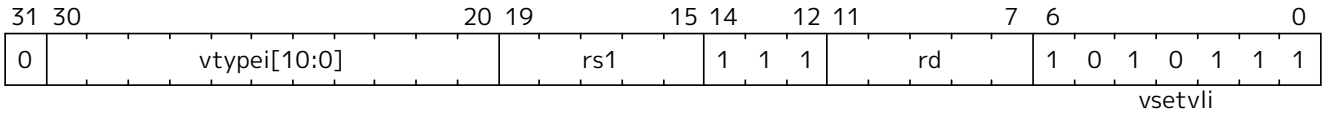


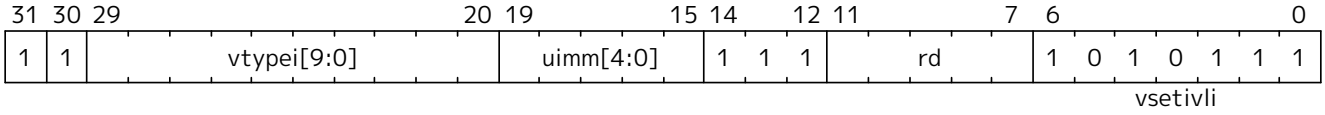
Formats for Vector Arithmetic Instructions under OP-V major opcode

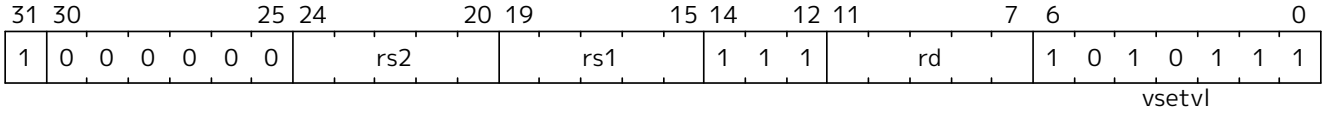




Formats for Vector Configuration Instructions under OP-V major opcode







Vector instructions can have scalar or vector source operands and produce scalar or vector results, and most vector instructions can be performed either unconditionally or conditionally under a mask.

Vector loads and stores move bit patterns between vector register elements and memory. Vector arithmetic instructions operate on values held in vector register elements.

31.5.1. Scalar Operands

Scalar operands can be immediates, or taken from the `x` registers, the `f` registers, or element 0 of a vector register. Scalar results are written to an `x` or `f` register or to element 0 of a vector register. Any vector register can be used to hold a scalar regardless of the current LMUL setting.

Zfinx ("F in X") is a new ISA extension where floating-point instructions take their arguments from the integer register file. The vector extension is also compatible with Zfinx, where the Zfinx vector extension has vector-scalar floating-point instructions taking their scalar argument from the `x` *registers.*

We considered but did not pursue overlaying the `f` *registers on* `v` *registers. The adopted approach reduces vector register pressure, avoids interactions with the standard calling convention, simplifies high-performance scalar floating-point design, and provides compatibility with the Zfinx ISA option. Overlaying* `f` *with* `v` *would provide the advantage of lowering the number of state bits in some implementations, but complicates high-performance designs and would prevent compatibility with the Zfinx ISA option.*

31.5.2. Vector Operands

Each vector operand has an *effective element width* (EEW) and an *effective* LMUL (EMUL) that is used to determine the size and location of all the elements within a vector register group. By default, for most operands of most instructions, EEW=SEW and EMUL=LMUL.

Some vector instructions have source and destination vector operands with the same number of elements but different widths, so that EEW and EMUL differ from SEW and LMUL respectively but EEW/EMUL = SEW/LMUL. For example, most widening arithmetic instructions have a source group with EEW=SEW and EMUL=LMUL but have a destination group with EEW=2*SEW and EMUL=2*LMUL. Narrowing instructions have a source operand that has EEW=2*SEW and EMUL=2*LMUL but with a destination

where EEW=SEW and EMUL=LMUL.

Vector operands or results may occupy one or more vector registers depending on EMUL, but are always specified using the lowest-numbered vector register in the group. Using other than the lowest-numbered vector register to specify a vector register group is a reserved encoding.

A vector register cannot be used to provide source operands with more than one EEW for a single instruction. A mask register source is considered to have EEW=1 for this constraint. An encoding that would result in the same vector register being read with two or more different EEWs, including when the vector register appears at different positions within two or more vector register groups, is reserved.

In practice, there is no software benefit to reading the same register with different EEW in the same instruction, and this constraint reduces complexity for implementations that internally rearrange data dependent on EEW.

A destination vector register group can overlap a source vector register group only if one of the following holds:

- The destination EEW equals the source EEW.
- The destination EEW is smaller than the source EEW, and the lowest-numbered register in the destination vector register group is the same as the lowest-numbered register in the source vector register group. (For example, when LMUL=1, `vnsrl.wi v0, v0, 3` is legal, but a destination of `v1` is not).
- The destination EEW is greater than the source EEW, the source EMUL is at least 1, and the highest-numbered register in the destination vector register group is the same as the highest-numbered register in the source vector register group. (For example, when LMUL=8, `vzext.vf4 v0, v6` is legal, but a source of `v0`, `v2`, or `v4` is not).

For the purpose of determining register group overlap constraints, mask elements have EEW=1.

The overlap constraints are designed to support resumable exceptions in machines without register renaming.

Any instruction encoding that violates the overlap constraints is reserved.

When source and destination registers overlap and have different EEW, the instruction is mask- and tail-agnostic, regardless of the setting of the `vta` and `vma` bits in `vtype`.

The largest vector register group used by an instruction can not be greater than 8 vector registers (i.e., EMUL≤8), and if a vector instruction would require greater than 8 vector registers in a group, the instruction encoding is reserved. For example, a widening operation that produces a widened vector register group result when LMUL=8 is reserved as this would imply a result EMUL=16.

Widened scalar values, e.g., input and output to a widening reduction operation, are held in the first element of a vector register and have EMUL=1.

31.5.3. Vector Masking

Masking is supported on many vector instructions. Element operations that are masked off (inactive) never generate exceptions. The destination vector register elements corresponding to masked-off elements are handled with either a mask-undisturbed or mask-agnostic policy depending on the setting of the `vma` bit in `vtype` (Section 31.3.4.3).

The mask value used to control execution of a masked vector instruction is always supplied by vector

register `v0`.

Masks are held in vector registers, rather than in a separate mask register file, to reduce total architectural state and to simplify the ISA.

Future vector extensions may provide longer instruction encodings with space for a full mask register specifier.

The destination vector register group for a masked vector instruction cannot overlap the source mask register (`v0`), unless the destination vector register is being written with a mask value (e.g., compares) or the scalar result of a reduction. These instruction encodings are reserved.

This constraint supports restart with a non-zero `vstart` *value.*

Other vector registers can be used to hold working mask values, and mask vector logical operations are provided to perform predicate calculations.

As specified in Section 31.3.4.3, mask destination tail elements are always treated as tail-agnostic, regardless of the setting of `vta`.

31.5.3.1. Mask Encoding

Where available, masking is encoded in a single-bit `vm` field in the instruction (`inst[25]`).

vm	Description
0	vector result, only where v0.mask[i] = 1
1	unmasked

Vector masking is represented in assembler code as another vector operand, with `.t` indicating that the operation occurs when `v0.mask[i]` is `1` (`t` for "true"). If no masking operand is specified, unmasked vector execution (`vm=1`) is assumed.

```
vop.v*    v1, v2, v3, v0.t  # enabled where v0.mask[i]=1, vm=0
vop.v*    v1, v2, v3        # unmasked vector operation, vm=1
```

Even though the current vector extensions only support one vector mask register `v0` *and only the true form of predication, the assembly syntax writes it out in full to be compatible with future extensions that might add a mask register specifier and support both true and complement mask values. The* `.t` *suffix on the masking operand also helps to visually encode the use of a mask.*

The `.mask` *suffix is not part of the assembly syntax. We only append it in contexts where a mask vector is subscripted, e.g.,* `v0.mask[i]`.

31.5.4. Prestart, Active, Inactive, Body, and Tail Element Definitions

The destination element indices operated on during a vector instruction's execution can be divided into three disjoint subsets.

- The *prestart* elements are those whose element index is less than the initial value in the `vstart` register. The prestart elements do not raise exceptions and do not update the destination vector register.

- The *body* elements are those whose element index is greater than or equal to the initial value in the `vstart` register, and less than the current vector length setting in `vl`. The body can be split into two disjoint subsets:
  - The *active* elements during a vector instruction's execution are the elements within the body and where the current mask is enabled at that element position. The active elements can raise exceptions and update the destination vector register group.
  - The *inactive* elements are the elements within the body but where the current mask is disabled at that element position. The inactive elements do not raise exceptions and do not update any destination vector register group unless masked agnostic is specified (`vtype.vma`=1), in which case inactive elements may be overwritten with 1s.
- The *tail* elements during a vector instruction's execution are the elements past the current vector length setting specified in `vl`. The tail elements do not raise exceptions, and do not update any destination vector register group unless tail agnostic is specified (`vtype.vta`=1), in which case tail elements may be overwritten with 1s, or with the result of the instruction in the case of mask-producing instructions except for mask loads. When LMUL < 1, the tail includes the elements past VLMAX that are held in the same vector register.

```
for element index x
prestart(x) = (0 <= x < vstart)
body(x)     = (vstart <= x < vl)
tail(x)     = (vl <= x < max(VLMAX,VLEN/SEW))
mask(x)     = unmasked || v0.mask[x] == 1
active(x)   = body(x) && mask(x)
inactive(x) = body(x) && !mask(x)
```

When `vstart` ≥ `vl`, there are no body elements, and no elements are updated in any destination vector register group, including that no tail elements are updated with agnostic values.

As a consequence, when `vl`*=0, no elements, including agnostic elements, are updated in the destination vector register group regardless of* `vstart`.

Instructions that write an `x` register or `f` register do so even when `vstart` ≥ `vl`, including when `vl`=0.

Some instructions such as `vslidedown` *and* `vrgather` *may read indices past* `vl` *or even VLMAX in source vector register groups. The general policy is to return the value 0 when the index is greater than VLMAX in the source vector register group.*

31.6. Configuration-Setting Instructions (vsetvli/vsetivli/vsetvl)

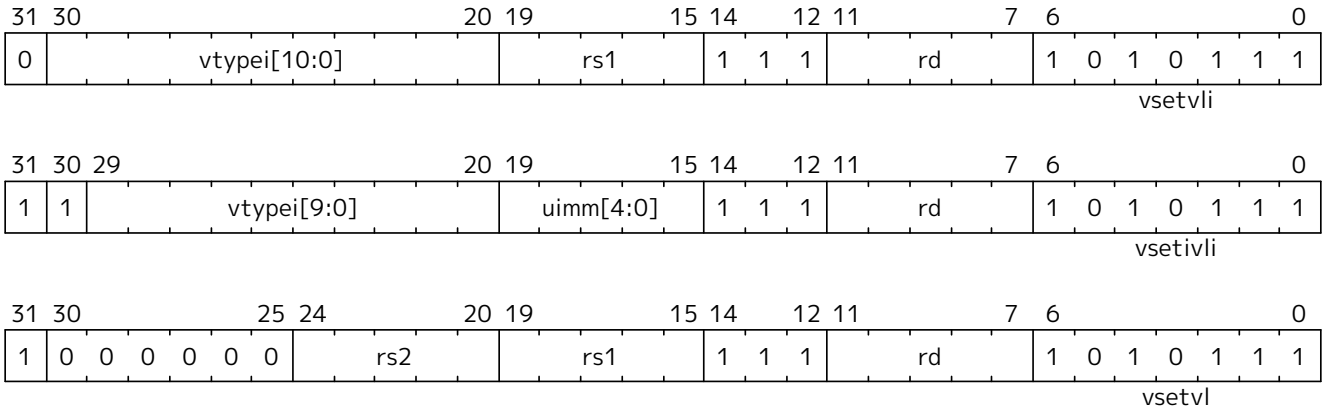
One of the common approaches to handling a large number of elements is "strip mining" where each iteration of a loop handles some number of elements, and the iterations continue until all elements have been processed. The RISC-V vector specification provides direct, portable support for this approach. The application specifies the total number of elements to be processed (the application vector length or AVL) as a candidate value for `vl`, and the hardware responds via a general-purpose register with the (frequently smaller) number of elements that the hardware will handle per iteration (stored in `vl`), based on the microarchitectural implementation and the `vtype` setting. A straightforward loop structure, shown in Section 31.6.4, depicts the ease with which the code keeps track of the remaining number of elements and the amount per iteration handled by hardware.

A set of instructions is provided to allow rapid configuration of the values in `vl` and `vtype` to match

application needs. The `vset{i}vl{i}` instructions set the `vtype` and `vl` CSRs based on their arguments, and write the new value of `vl` into `rd`.

```
 vsetvli rd, rs1, vtypei   # rd = new vl, rs1 = AVL, vtypei = new vtype setting
 vsetivli rd, uimm, vtypei # rd = new vl, uimm = AVL, vtypei = new vtype
setting
 vsetvl  rd, rs1, rs2      # rd = new vl, rs1 = AVL, rs2 = new vtype value
```

Formats for Vector Configuration Instructions under OP-V major opcode



31.6.1. `vtype` encoding

This diagram shows the layout for RV32 systems, whereas in general `vill` *should be at bit XLEN-1.*

Table 48. `vtype` *register layout*

Bits	Name	Description
XLEN-1	vill	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	vma	Vector mask agnostic
6	vta	Vector tail agnostic
5:3	vsew[2:0]	Selected element width (SEW) setting
2:0	vlmul[2:0]	Vector register group multiplier (LMUL) setting

The new `vtype` value is encoded in the immediate fields of `vsetvli` and `vsetivli`, and in the `rs2` register for `vsetvl`.

```
Suggested assembler names used for vset{i}vli vtypei immediate

e8    # SEW=8b
e16   # SEW=16b
e32   # SEW=32b
e64   # SEW=64b
```

```
 mf8  # LMUL=1/8
 mf4  # LMUL=1/4
 mf2  # LMUL=1/2
 m1   # LMUL=1
 m2   # LMUL=2
 m4   # LMUL=4
 m8   # LMUL=8

Examples:
    vsetvli t0, a0, e8, m1, ta, ma      # SEW= 8, LMUL=1
    vsetvli t0, a0, e8, m2, ta, ma      # SEW= 8, LMUL=2
    vsetvli t0, a0, e32, mf2, ta, ma    # SEW=32, LMUL=1/2
```

The `vsetvl` variant operates similarly to `vsetvli` except that it takes a `vtype` value from `rs2` and can be used for context restore.

31.6.1.1. Unsupported `vtype` Values

If the `vtype` value is not supported by the implementation, then the `vill` bit is set in `vtype`, the remaining bits in `vtype` are set to zero, and the `vl` register is also set to zero.

Earlier drafts required a trap when setting `vtype` to an illegal value. However, this would have added the first data-dependent trap on a CSR write to the ISA. Implementations could choose to trap when illegal values are written to `vtype` instead of setting `vill`, to allow emulation to support new configurations for forward-compatibility. The current scheme supports light-weight runtime interrogation of the supported vector unit configurations by checking if `vill` is clear for a given setting.

A `vtype` value with `vill` set is treated as an unsupported configuration.

Implementations must consider all bits of the `vtype` value to determine if the configuration is supported. An unsupported value in any location within the `vtype` value must result in `vill` being set.

In particular, all XLEN bits of the register `vtype` argument to the `vsetvl` instruction must be checked. Implementations cannot ignore fields they do not implement. All bits must be checked to ensure that new code assuming unsupported vector features in `vtype` traps instead of executing incorrectly on an older implementation.

31.6.2. AVL encoding

The new vector length setting is based on AVL, which for `vsetvli` and `vsetvl` is encoded in the `rs1` and `rd` fields as follows:

Table 49. AVL used in `vsetvli` and `vsetvl` instructions

`rd`	`rs1`	AVL value	Effect on `vl`
-	!x0	Value in `x[rs1]`	Normal strip mining
!x0	x0	~0	Set `vl` to VLMAX
x0	x0	Value in `vl` register	Keep existing `vl` (of course, `vtype` may change)

When *rs1* is not `x0`, the AVL is an unsigned integer held in the `x` register specified by *rs1*, and the new `vl` value is also written to the `x` register specified by *rd*.

When *rs1*=`x0` but *rd*≠`x0`, the maximum unsigned integer value (`~0`) is used as the AVL, and the resulting VLMAX is written to `vl` and also to the `x` register specified by `rd`.

When *rs1*=`x0` and *rd*=`x0`, the instructions operate as if the current vector length in `vl` is used as the AVL, and the resulting value is written to `vl`, but not to a destination register. This form can only be used when VLMAX and hence `vl` is not actually changed by the new SEW/LMUL ratio. Use of the instructions with a new SEW/LMUL ratio that would result in a change of VLMAX is reserved. Use of the instructions is also reserved if `vill` was 1 beforehand. Implementations may set `vill` in either case.

This last form of the instructions allows the `vtype` register to be changed while maintaining the current `vl`, provided VLMAX is not reduced. This design was chosen to ensure `vl` would always hold a legal value for current `vtype` setting. The current `vl` value can be read from the `vl` CSR. The `vl` value could be reduced by these instructions if the new SEW/LMUL ratio causes VLMAX to shrink, and so this case has been reserved as it is not clear this is a generally useful operation, and implementations can otherwise assume `vl` is not changed by these instructions to optimize their microarchitecture.

For the `vsetivli` instruction, the AVL is encoded as a 5-bit zero-extended immediate (0—31) in the `rs1` field.

The encoding of AVL for `vsetivli` is the same as for regular CSR immediate values.

The `vsetivli` instruction provides more compact code when the dimensions of vectors are small and known to fit inside the vector registers, in which case there is no strip-mining overhead.

31.6.3. Constraints on Setting `vl`

The `vset{i}vl{i}` instructions first set VLMAX according to their `vtype` argument, then set `vl` obeying the following constraints:

1. `vl = AVL` if `AVL ≤ VLMAX`
2. `ceil(AVL / 2) ≤ vl ≤ VLMAX` if `AVL < (2 * VLMAX)`
3. `vl = VLMAX` if `AVL ≥ (2 * VLMAX)`
4. Deterministic on any given implementation for same input AVL and VLMAX values
5. These specific properties follow from the prior rules:
   a. `vl = 0` if `AVL = 0`
   b. `vl > 0` if `AVL > 0`
   c. `vl ≤ VLMAX`
   d. `vl ≤ AVL`
   e. a value read from `vl` when used as the AVL argument to `vset{i}vl{i}` results in the same value in `vl`, provided the resultant VLMAX equals the value of VLMAX at the time that `vl` was read

The `vl` setting rules are designed to be sufficiently strict to preserve `vl` behavior across register spills and context swaps for `AVL ≤ VLMAX`, yet flexible enough to enable implementations to improve vector lane utilization for `AVL > VLMAX`.

For example, this permits an implementation to set `vl = ceil(AVL / 2)` for `VLMAX < AVL <`

> `2*VLMAX` *in order to evenly distribute work over the last two iterations of a strip-mine loop. Requirement 2 ensures that the first strip-mine iteration of reduction loops uses the largest vector length of all iterations, even in the case of* `AVL < 2*VLMAX`*. This allows software to avoid needing to explicitly calculate a running maximum of vector lengths observed during a strip-mined loop. Requirement 2 also allows an implementation to set vl to VLMAX for* `VLMAX < AVL < 2*VLMAX`

31.6.4. Example of strip mining and changes to SEW

The SEW and LMUL settings can be changed dynamically to provide high throughput on mixed-width operations in a single loop.

```
# Example: Load 16-bit values, widen multiply to 32b, shift 32b result
# right by 3, store 32b values.
# On entry:
#  a0 holds the total number of elements to process
#  a1 holds the address of the source array
#  a2 holds the address of the destination array

loop:
    vsetvli a3, a0, e16, m4, ta, ma  # vtype = 16-bit integer vectors;
                                     # also update a3 with vl (# of elements
this iteration)
    vle16.v v4, (a1)        # Get 16b vector
    slli t1, a3, 1          # Multiply # elements this iteration by 2
bytes/source element
    add a1, a1, t1          # Bump pointer
    vwmul.vx v8, v4, x10    # Widening multiply into 32b in <v8--v15>

    vsetvli x0, x0, e32, m8, ta, ma  # Operate on 32b values
    vsrl.vi v8, v8, 3
    vse32.v v8, (a2)        # Store vector of 32b elements
    slli t1, a3, 2          # Multiply # elements this iteration by 4
bytes/destination element
    add a2, a2, t1          # Bump pointer
    sub a0, a0, a3          # Decrement count by vl
    bnez a0, loop           # Any more?
```

31.7. Vector Loads and Stores

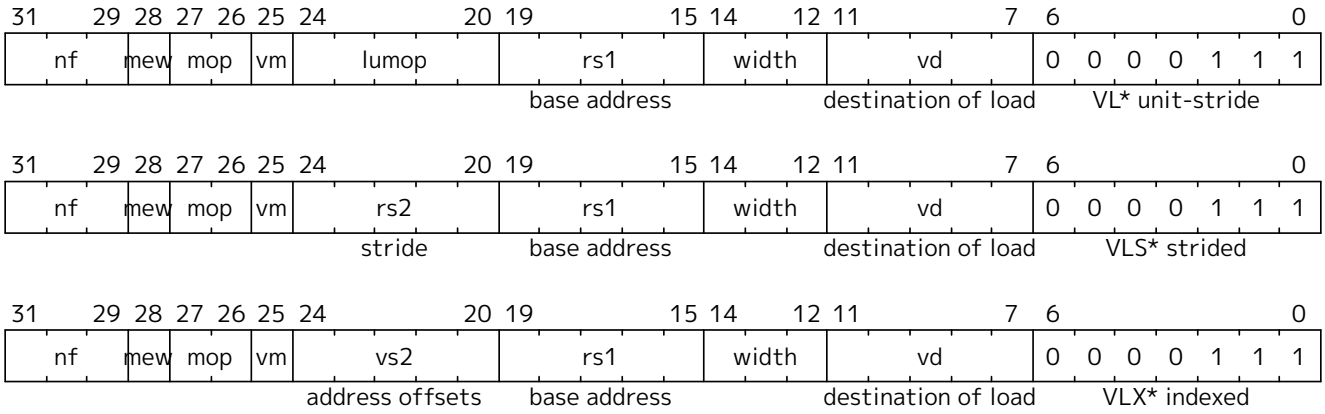
Vector loads and stores move values between vector registers and memory. Vector loads and stores can be masked, and they only access memory or raise exceptions for active elements. Masked vector loads do not update inactive elements in the destination vector register group, unless masked agnostic is specified (`vtype.vma`=1).

All vector loads and stores may generate and accept a non-zero `vstart` value.

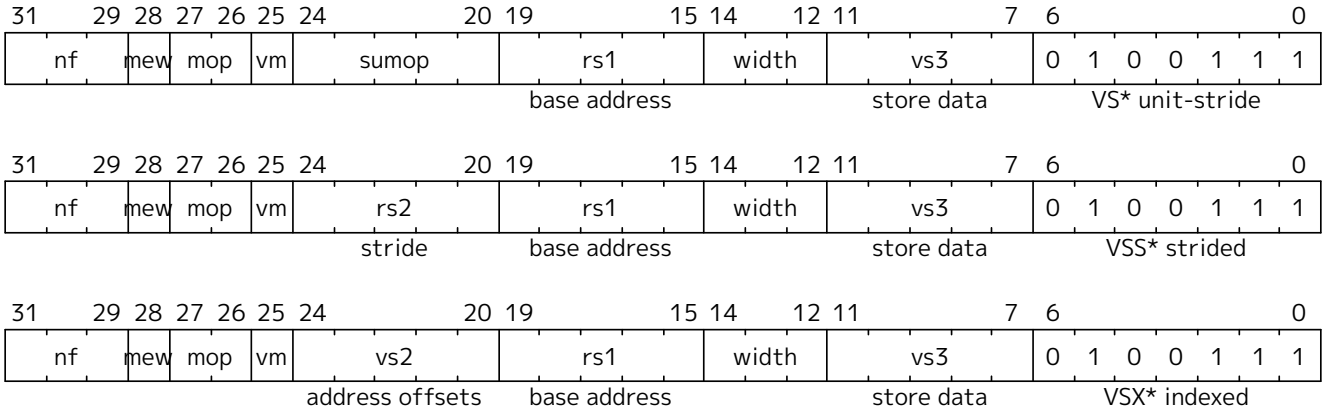
31.7.1. Vector Load/Store Instruction Encoding

Vector loads and stores are encoded within the scalar floating-point load and store major opcodes (LOAD-FP/STORE-FP). The vector load and store encodings repurpose a portion of the standard scalar floating-point load/store 12-bit immediate field to provide further vector instruction encoding, with bit 25 holding the standard vector mask bit (see Section 31.5.3.1).

Format for Vector Load Instructions under LOAD-FP major opcode



Format for Vector Store Instructions under STORE-FP major opcode



Field	Description
rs1[4:0]	specifies x register holding base address
rs2[4:0]	specifies x register holding stride
vs2[4:0]	specifies v register holding address offsets
vs3[4:0]	specifies v register holding store data
vd[4:0]	specifies v register destination of load
vm	specifies whether vector masking is enabled (0 = mask enabled, 1 = mask disabled)
width[2:0]	specifies size of memory elements, and distinguishes from FP scalar
mew	extended memory element width. See Section 31.7.3
mop[1:0]	specifies memory addressing mode
nf[2:0]	specifies the number of fields in each segment, for segment load/stores
lumop[4:0]/sumop[4:0]	are additional fields encoding variants of unit-stride instructions

Vector memory unit-stride and constant-stride operations directly encode EEW of the data to be transferred statically in the instruction to reduce the number of `vtype` changes when accessing memory in a mixed-width routine. Indexed operations use the explicit EEW encoding in the instruction to set the size

of the indices used, and use SEW/LMUL to specify the data width.

31.7.2. Vector Load/Store Addressing Modes

The vector extension supports unit-stride, constant-stride, and indexed (scatter/gather) addressing modes. Vector load/store base registers and strides are taken from the GPR `x` registers.

The base effective address for all vector accesses is given by the contents of the `x` register named in `rs1`.

Vector unit-stride operations access elements stored contiguously in memory starting from the base effective address.

Vector constant-stride operations access the first memory element at the base effective address, and then access subsequent elements at address increments given by the byte offset contained in the `x` register specified by `rs2`.

Vector indexed operations add the contents of each element of the vector offset operand specified by `vs2` to the base effective address to give the effective address of each element. The data vector register group has EEW=SEW, EMUL=LMUL, while the offset vector register group has EEW encoded in the instruction and EMUL=(EEW/SEW)*LMUL.

The vector offset operand is treated as a vector of byte-address offsets.

The indexed operations can also be used to access fields within a vector of objects, where the `vs2` *vector holds pointers to the base of the objects and the scalar* `x` *register holds the offset of the member field in each object. Supporting this case is why the indexed operations were not defined to scale the element indices by the data EEW.*

If the vector offset elements are narrower than XLEN, they are zero-extended to XLEN before adding to the base effective address. If the vector offset elements are wider than XLEN, the least-significant XLEN bits are used in the address calculation.

If the implementation does not support the EEW of the offset elements, the instruction is reserved.

A profile may place an upper limit on the maximum supported index EEW (e.g., only up to XLEN) smaller than ELEN.

The vector addressing modes are encoded using the 2-bit `mop[1:0]` field.

Table 50. encoding for loads

mop [1:0]		Description	Opcodes
0	0	unit-stride	VLE<EEW>
0	1	indexed-unordered	VLUXEI<EEW>
1	0	constant-stride	VLSE<EEW>
1	1	indexed-ordered	VLOXEI<EEW>

Table 51. encoding for stores

mop [1:0]		Description	Opcodes
0	0	unit-stride	VSE<EEW>
0	1	indexed-unordered	VSUXEI<EEW>
1	0	constant-stride	VSSE<EEW>

mop [1:0]		Description	Opcodes
1	1	indexed-ordered	VSOXEI<EEW>

Vector unit-stride and constant-stride memory accesses do not guarantee ordering between individual element accesses. The vector indexed load and store memory operations have two forms, ordered and unordered. The indexed-ordered variants preserve element ordering on memory accesses.

For unordered instructions (`mop[1:0]`!=11) there is no guarantee on element access order. If the accesses are to a strongly ordered IO region, the element accesses can be initiated in any order.

To provide ordered vector accesses to a strongly ordered IO region, the ordered indexed instructions should be used.

For implementations with precise vector traps, exceptions on indexed-unordered stores must also be precise.

Additional unit-stride vector addressing modes are encoded using the 5-bit `lumop` and `sumop` fields in the unit-stride load and store instruction encodings respectively.

Table 52. lumop

lumop[4:0]					Description
0	0	0	0	0	unit-stride load
0	1	0	0	0	unit-stride, whole register load
0	1	0	1	1	unit-stride, mask load, EEW=8
1	0	0	0	0	unit-stride fault-only-first
x	x	x	x	x	other encodings reserved

Table 53. sumop

sumop[4:0]					Description
0	0	0	0	0	unit-stride store
0	1	0	0	0	unit-stride, whole register store
0	1	0	1	1	unit-stride, mask store, EEW=8
x	x	x	x	x	other encodings reserved

The `nf[2:0]` field encodes the number of fields in each segment. For regular vector loads and stores, `nf`=0, indicating that a single value is moved between a vector register group and memory at each element position. Larger values in the `nf` field are used to access multiple contiguous fields within a segment as described below in Section 31.7.8.

The `nf[2:0]` field also encodes the number of whole vector registers to transfer for the whole vector register load/store instructions.

31.7.3. Vector Load/Store Width Encoding

Vector loads and stores have an EEW encoded directly in the instruction. The corresponding EMUL is calculated as EMUL = (EEW/SEW)*LMUL. If the EMUL would be out of range (EMUL>8 or EMUL<1/8), the instruction encoding is reserved. The vector register groups must have legal register specifiers for the selected EMUL, otherwise the instruction encoding is reserved.

Vector unit-stride and constant-stride use the EEW/EMUL encoded in the instruction for the data values,

while vector indexed loads and stores use the EEW/EMUL encoded in the instruction for the index values and the SEW/LMUL encoded in `vtype` for the data values.

Vector loads and stores are encoded using width values that are not claimed by the standard scalar floating-point loads and stores.

Implementations must provide vector loads and stores with EEWs corresponding to all supported SEW settings. Vector load/store encodings for unsupported EEW widths are reserved.

Table 54. Width encoding for vector loads and stores.

	mew	width [2:0]			Mem bits	Data Reg bits	Index bits	Opcodes
Standard scalar FP	x	0	0	1	16	FLEN	-	FLH/FSH
Standard scalar FP	x	0	1	0	32	FLEN	-	FLW/FSW
Standard scalar FP	x	0	1	1	64	FLEN	-	FLD/FSD
Standard scalar FP	x	1	0	0	128	FLEN	-	FLQ/FSQ
Vector 8b element	0	0	0	0	8	8	-	VLxE8/VSxE8
Vector 16b element	0	1	0	1	16	16	-	VLxE16/VSxE16
Vector 32b element	0	1	1	0	32	32	-	VLxE32/VSxE32
Vector 64b element	0	1	1	1	64	64	-	VLxE64/VSxE64
Vector 8b index	0	0	0	0	SEW	SEW	8	VLxEI8/VSxEI8
Vector 16b index	0	1	0	1	SEW	SEW	16	VLxEI16/VSxEI16
Vector 32b index	0	1	1	0	SEW	SEW	32	VLxEI32/VSxEI32
Vector 64b index	0	1	1	1	SEW	SEW	64	VLxEI64/VSxEI64
Reserved	1	X	X	X	-	-	-	

Mem bits is the size of each element accessed in memory.

Data reg bits is the size of each data element accessed in register.

Index bits is the size of each index accessed in register.

The `mew` bit (`inst[28]`) when set is expected to be used to encode expanded memory sizes of 128 bits and above, but these encodings are currently reserved.

31.7.4. Vector Unit-Stride Instructions

```
# Vector unit-stride loads and stores

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vle8.v    vd, (rs1), vm  #    8-bit unit-stride load
vle16.v   vd, (rs1), vm  #   16-bit unit-stride load
```

```
vle32.v   vd, (rs1), vm  #   32-bit unit-stride load
vle64.v   vd, (rs1), vm  #   64-bit unit-stride load

# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vse8.v    vs3, (rs1), vm  #    8-bit unit-stride store
vse16.v   vs3, (rs1), vm  #   16-bit unit-stride store
vse32.v   vs3, (rs1), vm  #   32-bit unit-stride store
vse64.v   vs3, (rs1), vm  #   64-bit unit-stride store
```

Additional unit-stride mask load and store instructions are provided to transfer mask values to/from memory. These operate similarly to unmasked byte loads or stores (EEW=8), except that the effective vector length is `evl`=ceil(`vl`/8) (i.e. EMUL=1), and the destination register is always written with a tail-agnostic policy.

```
# Vector unit-stride mask load
vlm.v vd, (rs1)   #  Load byte vector of length ceil(vl/8)

# Vector unit-stride mask store
vsm.v vs3, (rs1)  #  Store byte vector of length ceil(vl/8)
```

`vlm.v` and `vsm.v` are encoded with the same `width[2:0]`=0 encoding as `vle8.v` and `vse8.v`, but are distinguished by different `lumop` and `sumop` encodings. Since `vlm.v` and `vsm.v` operate as byte loads and stores, `vstart` is in units of bytes for these instructions.

`vlm.v` and `vsm.v` respect the `vill` field in `vtype`, as they depend on `vtype` indirectly through its constraints on `vl`.

The previous assembler mnemonics `vle1.v` and `vse1.v` were confusing as length was handled differently for these instructions versus other element load/store instructions. To avoid software churn, these older assembly mnemonics are being retained as aliases.

The primary motivation to provide mask load and store is to support machines that internally rearrange data to reduce cross-datapath wiring. However, these instructions also provide a convenient mechanism to use packed bit vectors in memory as mask values, and also reduce the cost of mask spill/fill by reducing need to change `vl`.

31.7.5. Vector Constant-Stride Instructions

```
# Vector constant-stride loads and stores

# vd destination, rs1 base address, rs2 byte constant-stride
vlse8.v    vd, (rs1), rs2, vm  #    8-bit constant-stride load
vlse16.v   vd, (rs1), rs2, vm  #   16-bit constant-stride load
vlse32.v   vd, (rs1), rs2, vm  #   32-bit constant-stride load
vlse64.v   vd, (rs1), rs2, vm  #   64-bit constant-stride load

# vs3 store data, rs1 base address, rs2 byte constant-stride
vsse8.v    vs3, (rs1), rs2, vm  #    8-bit constant-stride store
vsse16.v   vs3, (rs1), rs2, vm  #   16-bit constant-stride store
```

```
vsse32.v   vs3, (rs1), rs2, vm  #   32-bit constant-stride store
vsse64.v   vs3, (rs1), rs2, vm  #   64-bit constant-stride store
```

Negative and zero strides are supported.

Element accesses within a constant-stride instruction are unordered with respect to each other.

When `rs2`=`x0`, then an implementation is allowed, but not required, to perform fewer memory operations than the number of active elements, and may perform different numbers of memory operations across different dynamic executions of the same static instruction.

Compilers must be aware to not use the `x0` form for rs2 when the immediate stride is `0` if the intent is to require all memory accesses are performed.

When `rs2!=x0` and the value of `x[rs2]=0`, the implementation must perform one memory access for each active element (but these accesses will not be ordered).

As with other architectural mandates, implementations must appear to perform each memory access. Microarchitectures are free to optimize away accesses that would not be observed by another agent, for example, in idempotent memory regions obeying RVWMO. For non-idempotent memory regions, where by definition each access can be observed by a device, the optimization would not be possible.

When repeating ordered vector accesses to the same memory address are required, then an ordered indexed operation can be used.

31.7.6. Vector Indexed Instructions

```
# Vector indexed loads and stores

# Vector indexed-unordered load instructions
# vd destination, rs1 base address, vs2 byte offsets
vluxei8.v    vd, (rs1), vs2, vm  # unordered  8-bit indexed load of SEW data
vluxei16.v   vd, (rs1), vs2, vm  # unordered 16-bit indexed load of SEW data
vluxei32.v   vd, (rs1), vs2, vm  # unordered 32-bit indexed load of SEW data
vluxei64.v   vd, (rs1), vs2, vm  # unordered 64-bit indexed load of SEW data

# Vector indexed-ordered load instructions
# vd destination, rs1 base address, vs2 byte offsets
vloxei8.v    vd, (rs1), vs2, vm  # ordered  8-bit indexed load of SEW data
vloxei16.v   vd, (rs1), vs2, vm  # ordered 16-bit indexed load of SEW data
vloxei32.v   vd, (rs1), vs2, vm  # ordered 32-bit indexed load of SEW data
vloxei64.v   vd, (rs1), vs2, vm  # ordered 64-bit indexed load of SEW data

# Vector indexed-unordered store instructions
# vs3 store data, rs1 base address, vs2 byte offsets
vsuxei8.v   vs3, (rs1), vs2, vm # unordered  8-bit indexed store of SEW data
vsuxei16.v  vs3, (rs1), vs2, vm # unordered 16-bit indexed store of SEW data
vsuxei32.v  vs3, (rs1), vs2, vm # unordered 32-bit indexed store of SEW data
vsuxei64.v  vs3, (rs1), vs2, vm # unordered 64-bit indexed store of SEW data
```

```
# Vector indexed-ordered store instructions
# vs3 store data, rs1 base address, vs2 byte offsets
vsoxei8.v    vs3, (rs1), vs2, vm  # ordered  8-bit indexed store of SEW data
vsoxei16.v   vs3, (rs1), vs2, vm  # ordered 16-bit indexed store of SEW data
vsoxei32.v   vs3, (rs1), vs2, vm  # ordered 32-bit indexed store of SEW data
vsoxei64.v   vs3, (rs1), vs2, vm  # ordered 64-bit indexed store of SEW data
```

The assembler syntax for indexed loads and stores uses `ei`*x instead of* `e`*x to indicate the statically encoded EEW is of the index not the data.*

The indexed operations mnemonics have a "U" or "O" to distinguish between unordered and ordered, while the other vector addressing modes have no character. While this is perhaps a little less consistent, this approach minimizes disruption to existing software, as VSXEI previously meant "ordered" - and the opcode can be retained as an alias during transition to help reduce software churn.

31.7.7. Unit-stride Fault-Only-First Loads

The unit-stride fault-only-first load instructions are used to vectorize loops with data-dependent exit conditions ("while" loops). These instructions execute as a regular load except that they will only take a trap caused by a synchronous exception on element 0. If element 0 raises an exception, `vl` is not modified, and the trap is taken. If an element > 0 raises an exception, the corresponding trap is not taken, and the vector length `vl` is reduced to the index of the element that would have raised an exception.

Load instructions may overwrite active destination vector register group elements past the element index at which the trap is reported. Similarly, fault-only-first load instructions may update active destination elements past the element that causes trimming of the vector length (but not past the original vector length). The values of these spurious updates do not have to correspond to the values in memory at the addressed memory locations. Non-idempotent memory locations can only be accessed when it is known the corresponding element load operation will not be restarted due to a trap or vector-length trimming.

```
# Vector unit-stride fault-only-first loads

# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
vle8ff.v    vd, (rs1), vm  #    8-bit unit-stride fault-only-first load
vle16ff.v   vd, (rs1), vm  #   16-bit unit-stride fault-only-first load
vle32ff.v   vd, (rs1), vm  #   32-bit unit-stride fault-only-first load
vle64ff.v   vd, (rs1), vm  #   64-bit unit-stride fault-only-first load
```

```
strlen example using unit-stride fault-only-first instruction

# size_t strlen(const char *str)
# a0 holds *str

strlen:
    mv a3, a0             # Save start
loop:
```

```
    vsetvli a1, x0, e8, m8, ta, ma  # Vector of bytes of maximum length
    vle8ff.v v8, (a3)      # Load bytes
    csrr a1, vl           # Get bytes read
    vmseq.vi v0, v8, 0    # Set v0[i] where v8[i] = 0
    vfirst.m a2, v0       # Find first set bit
    add a3, a3, a1        # Bump pointer
    bltz a2, loop         # Not found?

    add a0, a0, a1        # Sum start + bump
    add a3, a3, a2        # Add index
    sub a0, a3, a0        # Subtract start address+bump

    ret
```

There is a security concern with fault-on-first loads, as they can be used to probe for valid effective addresses. The unit-stride versions only allow probing a region immediately contiguous to a known region, and so reduce the security impact when used in unprivileged code. However, code running in S-mode can establish arbitrary page translations that allow probing of random guest physical addresses provided by a hypervisor. Constant-stride and scatter/gather fault-only-first instructions are not provided due to lack of encoding space, but they can also represent a larger security hole, allowing even unprivileged software to easily check multiple random pages for accessibility without experiencing a trap. This standard does not address possible security mitigations for fault-only-first instructions.

Even when an exception is not raised, implementations are permitted to process fewer than `vl` elements and reduce `vl` accordingly, but if `vstart`=0 and `vl`>0, then at least one element must be processed.

When the fault-only-first instruction takes a trap due to an interrupt, implementations should not reduce `vl` and should instead set a `vstart` value.

When the fault-only-first instruction would trigger a debug data-watchpoint trap on an element after the first, implementations should not reduce `vl` but instead should trigger the debug trap as otherwise the event might be lost.

31.7.8. Vector Load/Store Segment Instructions

The vector load/store segment instructions move multiple contiguous fields in memory to and from consecutively numbered vector registers.

The name "segment" reflects that the items moved are subarrays with homogeneous elements. These operations can be used to transpose arrays between memory and registers, and can support operations on "array-of-structures" datatypes by unpacking each field in a structure into a separate vector register.

The three-bit `nf` field in the vector instruction encoding is an unsigned integer that contains one less than the number of fields per segment, *NFIELDS*.

Table 55. NFIELDS Encoding

nf[2:0]			NFIELDS
0	0	0	1
0	0	1	2

nf[2:0]			NFIELDS
0	1	0	3
0	1	1	4
1	0	0	5
1	0	1	6
1	1	0	7
1	1	1	8

The EMUL setting must be such that EMUL * NFIELDS ≤ 8, otherwise the instruction encoding is reserved.

*The product ceil(EMUL) * NFIELDS represents the number of underlying vector registers that will be touched by a segmented load or store instruction. This constraint makes this total no larger than 1/4 of the architectural register file, and the same as for regular operations with EMUL=8.*

Each field will be held in successively numbered vector register groups. When EMUL>1, each field will occupy a vector register group held in multiple successively numbered vector registers, and the vector register group for each field must follow the usual vector register alignment constraints (e.g., when EMUL=2 and NFIELDS=4, each field's vector register group must start at an even vector register, but does not have to start at a multiple of 8 vector register number).

If the vector register numbers accessed by the segment load or store would increment past 31, then the instruction encoding is reserved.

This constraint is to help allow for forward-compatibility with a possible future longer instruction encoding that has more addressable vector registers.

The `vl` register gives the number of segments to move, which is equal to the number of elements transferred to each vector register group. Masking is also applied at the level of whole segments.

For segment loads and stores, the individual memory accesses used to access fields within each segment are unordered with respect to each other even for ordered indexed segment loads and stores.

The `vstart` value is in units of whole segments. If a trap occurs during access to a segment, it is implementation-defined whether a subset of the faulting segment's accesses are performed before the trap is taken.

31.7.8.1. Vector Unit-Stride Segment Loads and Stores

The vector unit-stride load and store segment instructions move packed contiguous segments into multiple destination vector register groups.

Where the segments hold structures with heterogeneous-sized fields, software can later unpack individual structure fields using additional instructions after the segment load brings data into the vector registers.

The assembler prefixes `vlseg`/`vsseg` are used for unit-stride segment loads and stores respectively.

```
# Format
# In this syntax, <nf> equals NFIELDS and is an integer in the range [2, 8].
vlseg<nf>e<eew>.v vd, (rs1), vm      # Unit-stride segment load template
vsseg<nf>e<eew>.v vs3, (rs1), vm     # Unit-stride segment store template
```

```
# Examples
vlseg8e8.v vd, (rs1), vm   # Load eight vector registers with eight byte
fields.

vsseg3e32.v vs3, (rs1), vm  # Store packed vector of 3*4-byte segments from
vs3,vs3+1,vs3+2 to memory
```

For loads, the `vd` register will hold the first field loaded from the segment. For stores, the `vs3` register is read to provide the first field to be stored to each segment.

```
# Example 1
# Memory structure holds packed RGB pixels (24-bit data structure, 8bpp)
vsetvli a1, t0, e8, m1, ta, ma
vlseg3e8.v v8, (a0), vm
# v8 holds the red pixels
# v9 holds the green pixels
# v10 holds the blue pixels

# Example 2
# Memory structure holds complex values, 32b for real and 32b for imaginary
vsetvli a1, t0, e32, m1, ta, ma
vlseg2e32.v v8, (a0), vm
# v8 holds real
# v9 holds imaginary
```

There are also fault-only-first versions of the unit-stride instructions.

```
# Template for vector fault-only-first unit-stride segment loads.
vlseg<nf>e<eew>ff.v vd, (rs1),  vm    # Unit-stride fault-only-first segment
loads
```

For fault-only-first segment loads, if an exception is detected partway through accessing the zeroth segment, the trap is taken. If an exception is detected partway through accessing a subsequent segment, `vl` is reduced to the index of that segment.

In both cases, it is implementation-defined whether a subset of the segment is loaded.

These instructions may overwrite destination vector register group elements past the point at which a trap is reported or past the point at which vector length is trimmed.

31.7.8.2. Vector Constant-Stride Segment Loads and Stores

Vector constant-stride segment loads and stores move contiguous segments where each segment is separated by the byte-stride offset given in the `rs2` GPR argument.

Negative and zero strides are supported.

```
# Format
vlsseg<nf>e<eew>.v vd, (rs1), rs2, vm          # Constant-stride segment loads
vssseg<nf>e<eew>.v vs3, (rs1), rs2, vm         # Constant-stride segment stores

# Examples
vsetvli a1, t0, e8, m1, ta, ma
vlsseg3e8.v v4, (x5), x6   # Load bytes at addresses x5+i*x6   into v4[i],
                           #  and bytes at addresses x5+i*x6+1 into v5[i],
                           #  and bytes at addresses x5+i*x6+2 into v6[i].

# Examples
vsetvli a1, t0, e32, m1, ta, ma
vssseg2e32.v v2, (x5), x6   # Store words from v2[i] to address x5+i*x6
                            #   and words from v3[i] to address x5+i*x6+4
```

Accesses to the fields within each segment can occur in any order, including the case where the byte stride is such that segments overlap in memory.

31.7.8.3. Vector Indexed Segment Loads and Stores

Vector indexed segment loads and stores move contiguous segments where each segment is located at an address given by adding the scalar base address in the `rs1` field to byte offsets in vector register `vs2`. Both ordered and unordered forms are provided, where the ordered forms access segments in element order. However, even for the ordered form, accesses to the fields within an individual segment are not ordered with respect to each other.

The data vector register group has EEW=SEW, EMUL=LMUL, while the index vector register group has EEW encoded in the instruction with EMUL=(EEW/SEW)*LMUL.

The EMUL * NFIELDS ≤ 8 constraint applies to the data vector register group.

```
# Format
vluxseg<nf>ei<eew>.v vd, (rs1), vs2, vm   # Indexed-unordered segment loads
vloxseg<nf>ei<eew>.v vd, (rs1), vs2, vm   # Indexed-ordered segment loads
vsuxseg<nf>ei<eew>.v vs3, (rs1), vs2, vm  # Indexed-unordered segment stores
vsoxseg<nf>ei<eew>.v vs3, (rs1), vs2, vm  # Indexed-ordered segment stores

# Examples
vsetvli a1, t0, e8, m1, ta, ma
vluxseg3ei8.v v4, (x5), v3   # Load bytes at addresses x5+v3[i]   into v4[i],
                                #  and bytes at addresses x5+v3[i]+1 into
v5[i],
                                #  and bytes at addresses x5+v3[i]+2 into
v6[i].

# Examples
vsetvli a1, t0, e32, m1, ta, ma
vsuxseg2ei32.v v2, (x5), v5   # Store words from v2[i] to address x5+v5[i]
```

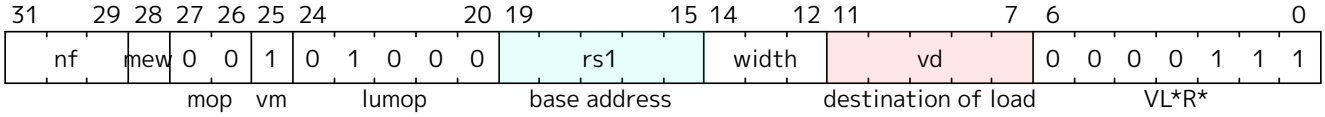
```
                                #   and words from v3[i] to address x5+v5[i]+4
```

For vector indexed segment loads, the destination vector register groups cannot overlap the source vector register group (specified by `vs2`), else the instruction encoding is reserved.

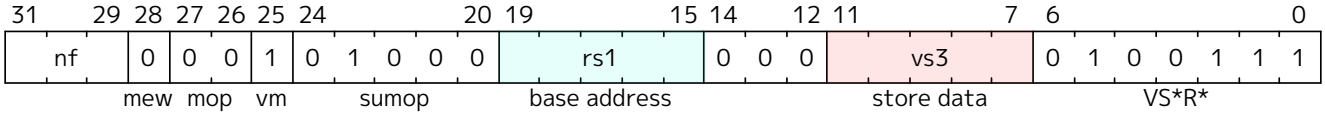
This constraint supports restart of indexed segment loads that raise exceptions partway through loading a structure.

31.7.9. Vector Load/Store Whole Register Instructions

Format for Vector Load Whole Register Instructions under LOAD-FP major opcode



Format for Vector Store Whole Register Instructions under STORE-FP major opcode



These instructions load and store whole vector register groups.

These instructions are intended to be used to save and restore vector registers when the type or length of the current contents of the vector register is not known, or where modifying `vl` and `vtype` would be costly. Examples include compiler register spills, vector function calls where values are passed in vector registers, interrupt handlers, and OS context switches. Software can determine the number of bytes transferred by reading the `vlenb` register.

The load instructions have an EEW encoded in the `mew` and `width` fields following the pattern of regular unit-stride loads.

Because in-register byte layouts are identical to in-memory byte layouts, the same data is written to the destination register group regardless of EEW. Hence, it would have sufficed to provide only EEW=8 variants. The full set of EEW variants is provided so that the encoded EEW can be used as a hint to indicate the destination register group will next be accessed with this EEW, which aids implementations that rearrange data internally.

The vector whole register store instructions are encoded similar to unmasked unit-stride store of elements with EEW=8.

The `nf` field encodes how many vector registers to load and store using the NFIELDS encoding (Figure Table 55). The encoded number of registers must be a power of 2 and the vector register numbers must be aligned as with a vector register group, otherwise the instruction encoding is reserved. NFIELDS indicates the number of vector registers to transfer, numbered successively after the base. Only NFIELDS values of 1, 2, 4, 8 are supported, with other values reserved. When multiple registers are transferred, the lowest-numbered vector register is held in the lowest-numbered memory addresses and successive vector register numbers are placed contiguously in memory.

The instructions operate with an effective vector length, `evl`=NFIELDS*VLEN/EEW, regardless of current settings in `vtype` and `vl`. The usual property that no elements are written if `vstart` ≥ `vl` does not apply to these instructions. Similarly, the property that the instructions are reserved if `vstart` exceeds the largest element index for the current `vtype` setting does not apply. Instead, the instructions are reserved if `vstart`

≥ `evl`.

The instructions operate similarly to unmasked unit-stride load and store instructions, with the base address passed in the scalar `x` register specified by `rs1`.

Implementations are allowed to raise a misaligned address exception on whole register loads and stores if the base address is not naturally aligned to the larger of the size of the encoded EEW in bytes (EEW/8) or the implementation's smallest supported SEW size in bytes (SEW_{MIN}/8).

Allowing misaligned exceptions to be raised based on non-alignment to the encoded EEW simplifies the implementation of these instructions. Some subset implementations might not support smaller SEW widths, so are allowed to report misaligned exceptions for the smallest supported SEW even if larger than encoded EEW. An extreme non-standard implementation might have SEW_{MIN}>XLEN for example. Software environments can mandate the minimum alignment requirements to support an ABI.

```
# Format of whole register load and store instructions.
vl1r.v v3, (a0)       # Pseudoinstruction equal to vl1re8.v

vl1re8.v    v3, (a0)  # Load v3 with VLEN/8 bytes held at address in a0
vl1re16.v   v3, (a0)  # Load v3 with VLEN/16 halfwords held at address in a0
vl1re32.v   v3, (a0)  # Load v3 with VLEN/32 words held at address in a0
vl1re64.v   v3, (a0)  # Load v3 with VLEN/64 doublewords held at address in a0

vl2r.v v2, (a0)       # Pseudoinstruction equal to vl2re8.v

vl2re8.v    v2, (a0)  # Load v2-v3 with 2*VLEN/8 bytes from address in a0
vl2re16.v   v2, (a0)  # Load v2-v3 with 2*VLEN/16 halfwords held at address in
a0
vl2re32.v   v2, (a0)  # Load v2-v3 with 2*VLEN/32 words held at address in a0
vl2re64.v   v2, (a0)  # Load v2-v3 with 2*VLEN/64 doublewords held at address
in a0

vl4r.v v4, (a0)       # Pseudoinstruction equal to vl4re8.v

vl4re8.v    v4, (a0)  # Load v4-v7 with 4*VLEN/8 bytes from address in a0
vl4re16.v   v4, (a0)
vl4re32.v   v4, (a0)
vl4re64.v   v4, (a0)

vl8r.v v8, (a0)       # Pseudoinstruction equal to vl8re8.v

vl8re8.v    v8, (a0)  # Load v8-v15 with 8*VLEN/8 bytes from address in a0
vl8re16.v   v8, (a0)
vl8re32.v   v8, (a0)
vl8re64.v   v8, (a0)

vs1r.v v3, (a1)      # Store v3 to address in a1
vs2r.v v2, (a1)      # Store v2-v3 to address in a1
vs4r.v v4, (a1)      # Store v4-v7 to address in a1
```

```
vs8r.v v8, (a1)      # Store v8-v15 to address in a1
```

We have considered adding a whole register mask load instruction (`vl1rm.v`) but have decided to omit from initial extension. The primary purpose would be to inform the microarchitecture that the data will be used as a mask. The same effect can be achieved with the following code sequence, whose cost is at most four instructions. Of these, the first could likely be removed as `vl` is often already in a scalar register, and the last might already be present if the following vector instruction needs a new SEW/LMUL. So, in best case only two instructions (of which only one performs vector operations) are needed to synthesize the effect of the dedicated instruction:

```
csrr t0, vl                         # Save current vl (potentially not needed)
vsetvli t1, x0, e8, m8, ta, ma      # Maximum VLMAX
vlm.v v0, (a0)                      # Load mask register
vsetvli x0, t0, <new type>          # Restore vl (potentially already present)
```

31.8. Vector Memory Alignment Constraints

If an element accessed by a vector memory instruction is not naturally aligned to the size of the element, either the element is transferred successfully or an address-misaligned exception is raised on that element.

Support for misaligned vector memory accesses is independent of an implementation's support for misaligned scalar memory accesses.

An implementation may have neither, one, or both scalar and vector memory accesses support some or all misaligned accesses in hardware. A separate PMA should be defined to determine if vector misaligned accesses are supported in the associated address range.

Vector misaligned memory accesses follow the same rules for atomicity as scalar misaligned memory accesses.

31.9. Vector Memory Consistency Model

Vector memory instructions appear to execute in program order on the local hart.

Vector memory instructions follow RVWMO at the instruction level. If the Ztso extension is implemented, vector memory instructions additionally follow RVTSO at the instruction level.

Except for vector indexed-ordered loads and stores, element operations are unordered within the instruction.

Vector indexed-ordered loads and stores read and write elements from/to memory in element order respectively, obeying RVWMO at the element level.

Ztso only imposes RVTSO at the instruction level; intra-instruction ordering follows RVWMO regardless of whether Ztso is implemented.

More formal definitions required.

Instructions affected by the vector length register `vl` have a control dependency on `vl`, rather than a data dependency.

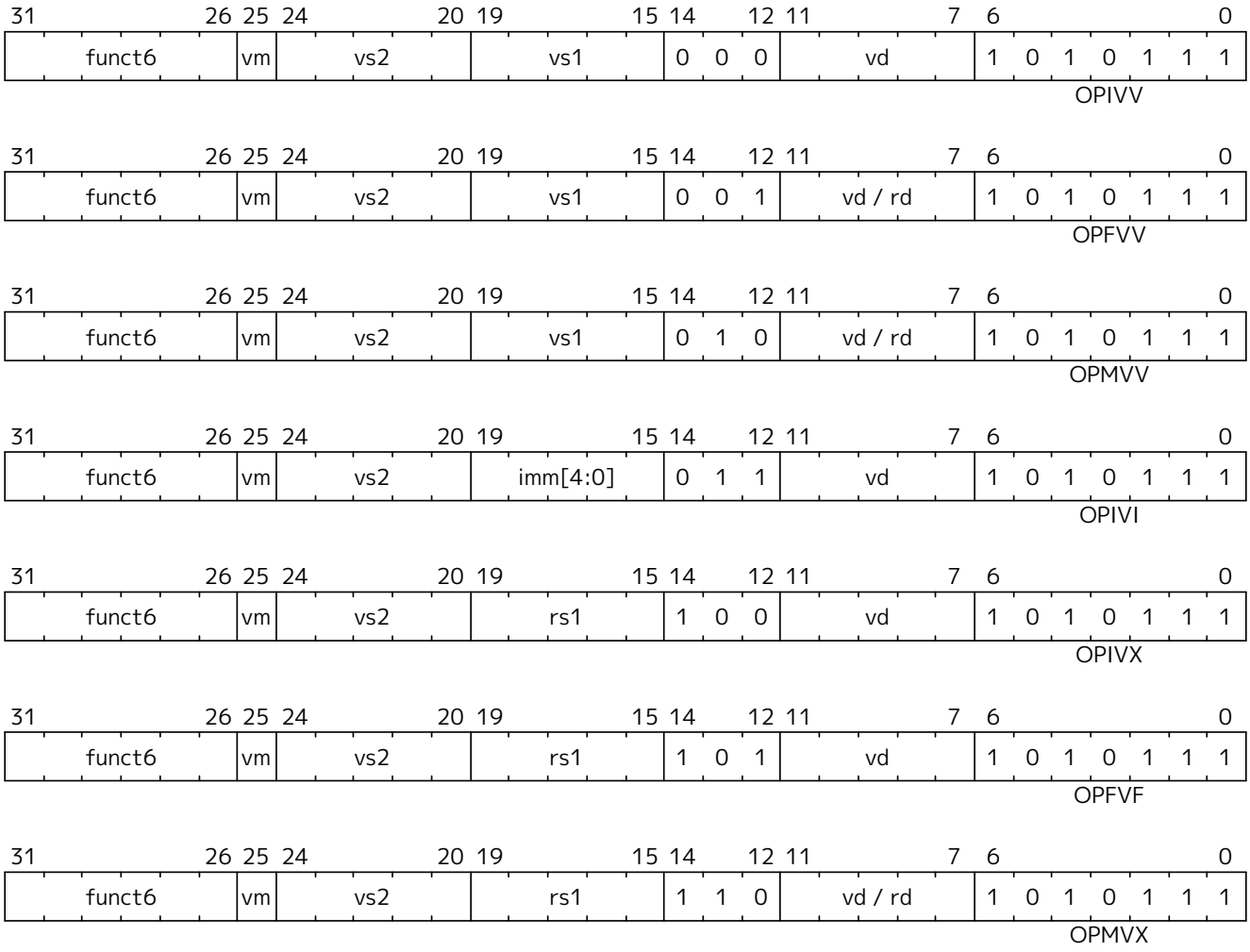
Similarly, masked vector instructions have a control dependency on the source mask register, rather than a data dependency.

Treating the vector length and mask as control rather than data typically matches the semantics of the corresponding scalar code, where branch instructions ordinarily would have been used. Treating the mask as control allows masked vector load instructions to access memory before the mask value is known, without the need for a misspeculation-recovery mechanism.

31.10. Vector Arithmetic Instruction Formats

The vector arithmetic instructions use a new major opcode (OP-V = 1010111_2) which neighbors OP-FP. The three-bit `funct3` field is used to define sub-categories of vector instructions.

Formats for Vector Arithmetic Instructions under OP-V major opcode



31.10.1. Vector Arithmetic Instruction encoding

The `funct3` field encodes the operand type and source locations.

Table 56. funct3

funct3[2:0]			Category	Operands	Type of scalar operand
0	0	0	OPIVV	vector-vector	N/A
0	0	1	OPFVV	vector-vector	N/A

funct3[2:0]			Category	Operands	Type of scalar operand
0	1	0	OPMVV	vector-vector	N/A
0	1	1	OPIVI	vector-immediate	`imm[4:0]`
1	0	0	OPIVX	vector-scalar	GPR `x` register `rs1`
1	0	1	OPFVF	vector-scalar	FP `f` register `rs1`
1	1	0	OPMVX	vector-scalar	GPR `x` register `rs1`
1	1	1	OPCFG	scalars-imms	GPR `x` register `rs1` & `rs2`/`imm`

Integer operations are performed using unsigned or two's-complement signed integer arithmetic depending on the opcode.

In this discussion, fixed-point operations are considered to be integer operations.

All standard vector floating-point arithmetic operations follow the IEEE-754/2008 standard. All vector floating-point operations use the dynamic rounding mode in the `frm` register. Use of the `frm` field when it contains an invalid rounding mode by any vector floating-point instruction—even those that do not depend on the rounding mode, or when `vl`=0, or when `vstart` ≥ `vl`--is reserved.

All vector floating-point code will rely on a valid value in `frm`. Implementations can make all vector FP instructions report exceptions when the rounding mode is invalid to simplify control logic.

Vector-vector operations take two vectors of operands from vector register groups specified by `vs2` and `vs1` respectively.

Vector-scalar operations can have three possible forms. In all three forms, the vector register group operand is specified by `vs2`. The second scalar source operand comes from one of three alternative sources:

1. For integer operations, the scalar can be a 5-bit immediate, `imm[4:0]`, encoded in the `rs1` field. The value is sign-extended to SEW bits, unless otherwise specified.
2. For integer operations, the scalar can be taken from the scalar `x` register specified by `rs1`. If XLEN>SEW, the least-significant SEW bits of the `x` register are used, unless otherwise specified. If XLEN<SEW, the value from the `x` register is sign-extended to SEW bits.
3. For floating-point operations, the scalar can be taken from a scalar `f` register. If FLEN > SEW, the value in the `f` registers is checked for a valid NaN-boxed value, in which case the least-significant SEW bits of the `f` register are used, else the canonical NaN value is used. Vector instructions where any floating-point vector operand's EEW is not a supported floating-point type width (which includes when FLEN < SEW) are reserved.

Some instructions zero-extend the 5-bit immediate, and denote this by naming the immediate `uimm` in the assembly syntax.

When adding a vector extension to the Zfinx/Zdinx/Zhinx extensions, floating-point scalar arguments are taken from the `x` registers. NaN-boxing is not supported in these extensions, and so operands narrower than XLEN bits are not checked for a NaN box; bits XLEN-1:EEW are ignored. For RV32_Zdinx, EEW=64 scalar arguments are supplied by an `x`-register pair.

Vector arithmetic instructions are masked under control of the `vm` field.

```
# Assembly syntax pattern for vector binary arithmetic instructions
```

```
# Operations returning vector results, masked by vm (v0.t, <nothing>)
vop.vv  vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op vs1[i]
vop.vx  vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op x[rs1]
vop.vi  vd, vs2, imm, vm  # integer vector-immediate   vd[i] = vs2[i] op imm

vfop.vv  vd, vs2, vs1, vm # FP vector-vector operation vd[i] = vs2[i] fop
vs1[i]
vfop.vf  vd, vs2, rs1, vm # FP vector-scalar operation vd[i] = vs2[i] fop
f[rs1]
```

In the encoding, `vs2` is the first operand, while `rs1/imm` is the second operand. This is the opposite to the standard scalar ordering. This arrangement retains the existing encoding conventions that instructions that read only one scalar register, read it from `rs1`, and that 5-bit immediates are sourced from the `rs1` field.

```
# Assembly syntax pattern for vector ternary arithmetic instructions (multiply-
add)

# Integer operations overwriting sum input
vop.vv vd, vs1, vs2, vm  # vd[i] = vs1[i] * vs2[i] + vd[i]
vop.vx vd, rs1, vs2, vm  # vd[i] = x[rs1] * vs2[i] + vd[i]

# Integer operations overwriting product input
vop.vv vd, vs1, vs2, vm  # vd[i] = vs1[i] * vd[i] + vs2[i]
vop.vx vd, rs1, vs2, vm  # vd[i] = x[rs1] * vd[i] + vs2[i]

# Floating-point operations overwriting sum input
vfop.vv vd, vs1, vs2, vm  # vd[i] = vs1[i] * vs2[i] + vd[i]
vfop.vf vd, rs1, vs2, vm  # vd[i] = f[rs1] * vs2[i] + vd[i]

# Floating-point operations overwriting product input
vfop.vv vd, vs1, vs2, vm  # vd[i] = vs1[i] * vd[i] + vs2[i]
vfop.vf vd, rs1, vs2, vm  # vd[i] = f[rs1] * vd[i] + vs2[i]
```

For ternary multiply-add operations, the assembler syntax always places the destination vector register first, followed by either `rs1` or `vs1`, then `vs2`. This ordering provides a more natural reading of the assembler for these ternary operations, as the multiply operands are always next to each other.

31.10.2. Widening Vector Arithmetic Instructions

A few vector arithmetic instructions are defined to be *widening* operations where the destination vector register group has EEW=2*SEW and EMUL=2*LMUL. These are generally given a `vw*` prefix on the opcode, or `vfw*` for vector floating-point instructions.

The first vector register group operand can be either single or double-width.

```
# Assembly syntax pattern for vector widening arithmetic instructions
```

```
# Double-width result, two single-width sources: 2*SEW = SEW op SEW
vwop.vv  vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op
vs1[i]
vwop.vx  vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op
x[rs1]

# Double-width result, first source double-width, second source single-width:
2*SEW = 2*SEW op SEW
vwop.wv  vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op
vs1[i]
vwop.wx  vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op
x[rs1]
```

Originally, a `w` *suffix was used on opcode, but this could be confused with the use of a* `w` *suffix to mean word-sized operations in doubleword integers, so the* `w` *was moved to prefix.*

The floating-point widening operations were changed to `vfw*` *from* `vwf*` *to be more consistent with any scalar widening floating-point operations that will be written as* `fw*`.

Widening instruction encodings must follow the constraints in Section 31.5.2.

31.10.3. Narrowing Vector Arithmetic Instructions

A few instructions are provided to convert double-width source vectors into single-width destination vectors. These instructions convert a vector register group specified by `vs2` with EEW/EMUL=2*SEW/2*LMUL to a vector register group with the current SEW/LMUL setting. Where there is a second source vector register group (specified by `vs1`), this has the same (narrower) width as the result (i.e., EEW=SEW).

An alternative design decision would have been to treat SEW/LMUL as defining the size of the source vector register group. The choice here is motivated by the belief the chosen approach will require fewer `vtype` *changes.*

Compare operations that set a mask register are also implicitly a narrowing operation.

A `vn*` prefix on the opcode is used to distinguish these instructions in the assembler, or a `vfn*` prefix for narrowing floating-point opcodes. The double-width source vector register group is signified by a `w` in the source operand suffix (e.g., `vnsra.wv`)

```
Assembly syntax pattern for vector narrowing arithmetic instructions

# Single-width result vd, double-width source vs2, single-width source vs1/rs1
# SEW = 2*SEW op SEW
vnop.wv  vd, vs2, vs1, vm  # integer vector-vector      vd[i] = vs2[i] op
vs1[i]
vnop.wx  vd, vs2, rs1, vm  # integer vector-scalar      vd[i] = vs2[i] op
x[rs1]
```

Narrowing instruction encodings must follow the constraints in Section 31.5.2.

31.11. Vector Integer Arithmetic Instructions

A set of vector integer arithmetic instructions is provided. Unless otherwise stated, integer operations wrap around on overflow.

31.11.1. Vector Single-Width Integer Add and Subtract

Vector integer add and subtract are provided. Reverse-subtract instructions are also provided for the vector-scalar forms.

```
# Integer adds.
vadd.vv vd, vs2, vs1, vm   # Vector-vector
vadd.vx vd, vs2, rs1, vm   # vector-scalar
vadd.vi vd, vs2, imm, vm   # vector-immediate

# Integer subtract
vsub.vv vd, vs2, vs1, vm   # Vector-vector
vsub.vx vd, vs2, rs1, vm   # vector-scalar

# Integer reverse subtract
vrsub.vx vd, vs2, rs1, vm   # vd[i] = x[rs1] - vs2[i]
vrsub.vi vd, vs2, imm, vm   # vd[i] = imm - vs2[i]
```

A vector of integer values can be negated using a reverse-subtract instruction with a scalar operand of `x0`*. An assembly pseudoinstruction* `vneg.v vd,vs` = `vrsub.vx vd,vs,x0` *is provided.*

31.11.2. Vector Widening Integer Add/Subtract

The widening add/subtract instructions are provided in both signed and unsigned variants, depending on whether the narrower source operands are first sign- or zero-extended before forming the double-width sum.

```
# Widening unsigned integer add/subtract, 2*SEW = SEW +/- SEW
vwaddu.vv  vd, vs2, vs1, vm  # vector-vector
vwaddu.vx  vd, vs2, rs1, vm  # vector-scalar
vwsubu.vv  vd, vs2, vs1, vm  # vector-vector
vwsubu.vx  vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = SEW +/- SEW
vwadd.vv  vd, vs2, vs1, vm  # vector-vector
vwadd.vx  vd, vs2, rs1, vm  # vector-scalar
vwsub.vv  vd, vs2, vs1, vm  # vector-vector
vwsub.vx  vd, vs2, rs1, vm  # vector-scalar

# Widening unsigned integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwaddu.wv  vd, vs2, vs1, vm  # vector-vector
vwaddu.wx  vd, vs2, rs1, vm  # vector-scalar
```

```
vwsubu.wv  vd, vs2, vs1, vm  # vector-vector
vwsubu.wx  vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwadd.wv  vd, vs2, vs1, vm  # vector-vector
vwadd.wx  vd, vs2, rs1, vm  # vector-scalar
vwsub.wv  vd, vs2, vs1, vm  # vector-vector
vwsub.wx  vd, vs2, rs1, vm  # vector-scalar
```

An integer value can be doubled in width using the widening add instructions with a scalar operand of `x0`*. Assembly pseudoinstructions* `vwcvt.x.x.v vd,vs,vm` = `vwadd.vx vd,vs,x0,vm` *and* `vwcvtu.x.x.v vd,vs,vm` = `vwaddu.vx vd,vs,x0,vm` *are provided.*

31.11.3. Vector Integer Extension

The vector integer extension instructions zero- or sign-extend a source vector integer operand with EEW less than SEW to fill SEW-sized elements in the destination. The EEW of the source is 1/2, 1/4, or 1/8 of SEW, while EMUL of the source is (EEW/SEW)*LMUL. The destination has EEW equal to SEW and EMUL equal to LMUL.

```
vzext.vf2 vd, vs2, vm  # Zero-extend SEW/2 source to SEW destination
vsext.vf2 vd, vs2, vm  # Sign-extend SEW/2 source to SEW destination
vzext.vf4 vd, vs2, vm  # Zero-extend SEW/4 source to SEW destination
vsext.vf4 vd, vs2, vm  # Sign-extend SEW/4 source to SEW destination
vzext.vf8 vd, vs2, vm  # Zero-extend SEW/8 source to SEW destination
vsext.vf8 vd, vs2, vm  # Sign-extend SEW/8 source to SEW destination
```

If the source EEW is not a supported width, or source EMUL would be below the minimum legal LMUL, the instruction encoding is reserved.

Standard vector load instructions access memory values that are the same size as the destination register elements. Some application code needs to operate on a range of operand widths in a wider element, for example, loading a byte from memory and adding to an eight-byte element. To avoid having to provide the cross-product of the number of vector load instructions by the number of data types (byte, word, halfword, and also signed/unsigned variants), we instead add explicit extension instructions that can be used if an appropriate widening arithmetic instruction is not available.

31.11.4. Vector Integer Add-with-Carry / Subtract-with-Borrow Instructions

To support multi-word integer arithmetic, instructions that operate on a carry bit are provided. For each operation (add or subtract), two instructions are provided: one to provide the result (SEW width), and the second to generate the carry output (single bit encoded as a mask boolean).

The carry inputs and outputs are represented using the mask register layout as described in Section 31.4.5. Due to encoding constraints, the carry input must come from the implicit `v0` register, but carry outputs can be written to any vector register that respects the source/destination overlap restrictions.

`vadc` and `vsbc` add or subtract the source operands and the carry-in or borrow-in, and write the result to vector register `vd`. These instructions are encoded as masked instructions (`vm=0`), but they operate on and

write back all body elements. Encodings corresponding to the unmasked versions (`vm=1`) are reserved.

`vmadc` and `vmsbc` add or subtract the source operands, optionally add the carry-in or subtract the borrow-in if masked (`vm=0`), and write the resulting carry-out or borrow-out back to mask register `vd`. If unmasked (`vm=1`), there is no carry-in or borrow-in. These instructions operate on and write back all body elements, even if masked. Because these instructions produce a mask value, they always operate with a tail-agnostic policy.

```
# Produce sum with carry.

# vd[i] = vs2[i] + vs1[i] + v0.mask[i]
vadc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = vs2[i] + x[rs1] + v0.mask[i]
vadc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd[i] = vs2[i] + imm + v0.mask[i]
vadc.vim   vd, vs2, imm, v0  # Vector-immediate

# Produce carry out in mask register format

# vd.mask[i] = carry_out(vs2[i] + vs1[i] + v0.mask[i])
vmadc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd.mask[i] = carry_out(vs2[i] + x[rs1] + v0.mask[i])
vmadc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd.mask[i] = carry_out(vs2[i] + imm + v0.mask[i])
vmadc.vim   vd, vs2, imm, v0  # Vector-immediate

# vd.mask[i] = carry_out(vs2[i] + vs1[i])
vmadc.vv    vd, vs2, vs1      # Vector-vector, no carry-in

# vd.mask[i] = carry_out(vs2[i] + x[rs1])
vmadc.vx    vd, vs2, rs1      # Vector-scalar, no carry-in

# vd.mask[i] = carry_out(vs2[i] + imm)
vmadc.vi    vd, vs2, imm      # Vector-immediate, no carry-in
```

Because implementing a carry propagation requires executing two instructions with unchanged inputs, destructive accumulations will require an additional move to obtain correct results.

```
# Example multi-word arithmetic sequence, accumulating into v4
vmadc.vvm v1, v4, v8, v0  # Get carry into temp register v1
vadc.vvm v4, v4, v8, v0   # Calc new sum
vmmv.m v0, v1             # Move temp carry into v0 for next word
```

The subtract with borrow instruction `vsbc` performs the equivalent function to support long word

arithmetic for subtraction. There are no subtract with immediate instructions.

```
# Produce difference with borrow.

# vd[i] = vs2[i] - vs1[i] - v0.mask[i]
vsbc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd[i] = vs2[i] - x[rs1] - v0.mask[i]
vsbc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# Produce borrow out in mask register format

# vd.mask[i] = borrow_out(vs2[i] - vs1[i] - v0.mask[i])
vmsbc.vvm   vd, vs2, vs1, v0  # Vector-vector

# vd.mask[i] = borrow_out(vs2[i] - x[rs1] - v0.mask[i])
vmsbc.vxm   vd, vs2, rs1, v0  # Vector-scalar

# vd.mask[i] = borrow_out(vs2[i] - vs1[i])
vmsbc.vv    vd, vs2, vs1      # Vector-vector, no borrow-in

# vd.mask[i] = borrow_out(vs2[i] - x[rs1])
vmsbc.vx    vd, vs2, rs1      # Vector-scalar, no borrow-in
```

For `vmsbc`, the borrow is defined to be 1 iff the difference, prior to truncation, is negative.

For `vadc` and `vsbc`, the instruction encoding is reserved if the destination vector register is `v0`.

This constraint corresponds to the constraint on masked vector operations that overwrite the mask register.

31.11.5. Vector Bitwise Logical Instructions

```
# Bitwise logical operations.
vand.vv vd, vs2, vs1, vm   # Vector-vector
vand.vx vd, vs2, rs1, vm   # vector-scalar
vand.vi vd, vs2, imm, vm   # vector-immediate

vor.vv vd, vs2, vs1, vm    # Vector-vector
vor.vx vd, vs2, rs1, vm    # vector-scalar
vor.vi vd, vs2, imm, vm    # vector-immediate

vxor.vv vd, vs2, vs1, vm    # Vector-vector
vxor.vx vd, vs2, rs1, vm    # vector-scalar
vxor.vi vd, vs2, imm, vm    # vector-immediate
```

With an immediate of -1, scalar-immediate forms of the `vxor` instruction provide a bitwise NOT operation. This is provided as an assembler pseudoinstruction `vnot.v vd,vs,vm` =

`vxor.vi vd,vs,-1,vm`.

31.11.6. Vector Single-Width Shift Instructions

A full set of vector shift instructions are provided, including logical shift left (`sll`), and logical (zero-extending `srl`) and arithmetic (sign-extending `sra`) shift right. The data to be shifted is in the vector register group specified by `vs2` and the shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. Only the low lg2(SEW) bits of the shift-amount value are used to control the shift amount.

```
# Bit shift operations
vsll.vv vd, vs2, vs1, vm   # Vector-vector
vsll.vx vd, vs2, rs1, vm   # vector-scalar
vsll.vi vd, vs2, uimm, vm   # vector-immediate

vsrl.vv vd, vs2, vs1, vm   # Vector-vector
vsrl.vx vd, vs2, rs1, vm   # vector-scalar
vsrl.vi vd, vs2, uimm, vm   # vector-immediate

vsra.vv vd, vs2, vs1, vm   # Vector-vector
vsra.vx vd, vs2, rs1, vm   # vector-scalar
vsra.vi vd, vs2, uimm, vm   # vector-immediate
```

31.11.7. Vector Narrowing Integer Right Shift Instructions

The narrowing right shifts extract a smaller field from a wider operand and have both zero-extending (`srl`) and sign-extending (`sra`) forms. The shift amount can come from a vector register group, or a scalar `x` register, or a zero-extended 5-bit immediate. The low lg2(2*SEW) bits of the shift-amount value are used (e.g., the low 6 bits for a SEW=64-bit to SEW=32-bit narrowing operation).

```
# Narrowing shift right logical, SEW = (2*SEW) >> SEW
vnsrl.wv vd, vs2, vs1, vm   # vector-vector
vnsrl.wx vd, vs2, rs1, vm   # vector-scalar
vnsrl.wi vd, vs2, uimm, vm   # vector-immediate

# Narrowing shift right arithmetic, SEW = (2*SEW) >> SEW
vnsra.wv vd, vs2, vs1, vm   # vector-vector
vnsra.wx vd, vs2, rs1, vm   # vector-scalar
vnsra.wi vd, vs2, uimm, vm   # vector-immediate
```

Future extensions might add support for versions that narrow to a destination that is 1/4 the width of the source.

An integer value can be halved in width using the narrowing integer shift instructions with a scalar operand of `x0`. *An assembly pseudoinstruction is provided* `vncvt.x.x.w vd,vs,vm` = `vnsrl.wx vd,vs,x0,vm`.

31.11.8. Vector Integer Compare Instructions

The following integer compare instructions write 1 to the destination mask register element if the comparison evaluates to true, and 0 otherwise. The destination mask vector is always held in a single vector register, with a layout of elements as described in Section 31.4.5. The destination mask vector register may be the same as the source vector mask register (`v0`).

```
# Set if equal
vmseq.vv vd, vs2, vs1, vm  # Vector-vector
vmseq.vx vd, vs2, rs1, vm  # vector-scalar
vmseq.vi vd, vs2, imm, vm  # vector-immediate

# Set if not equal
vmsne.vv vd, vs2, vs1, vm  # Vector-vector
vmsne.vx vd, vs2, rs1, vm  # vector-scalar
vmsne.vi vd, vs2, imm, vm  # vector-immediate

# Set if less than, unsigned
vmsltu.vv vd, vs2, vs1, vm  # Vector-vector
vmsltu.vx vd, vs2, rs1, vm  # Vector-scalar

# Set if less than, signed
vmslt.vv vd, vs2, vs1, vm  # Vector-vector
vmslt.vx vd, vs2, rs1, vm  # vector-scalar

# Set if less than or equal, unsigned
vmsleu.vv vd, vs2, vs1, vm   # Vector-vector
vmsleu.vx vd, vs2, rs1, vm   # vector-scalar
vmsleu.vi vd, vs2, imm, vm   # Vector-immediate

# Set if less than or equal, signed
vmsle.vv vd, vs2, vs1, vm  # Vector-vector
vmsle.vx vd, vs2, rs1, vm  # vector-scalar
vmsle.vi vd, vs2, imm, vm  # vector-immediate

# Set if greater than, unsigned
vmsgtu.vx vd, vs2, rs1, vm   # Vector-scalar
vmsgtu.vi vd, vs2, imm, vm   # Vector-immediate

# Set if greater than, signed
vmsgt.vx vd, vs2, rs1, vm    # Vector-scalar
vmsgt.vi vd, vs2, imm, vm    # Vector-immediate

# Following two instructions are not provided directly
# Set if greater than or equal, unsigned
# vmsgeu.vx vd, vs2, rs1, vm    # Vector-scalar
# Set if greater than or equal, signed
# vmsge.vx vd, vs2, rs1, vm    # Vector-scalar
```

The following table indicates how all comparisons are implemented in native machine code.

```
Comparison      Assembler Mapping             Assembler Pseudoinstruction

va < vb         vmslt{u}.vv vd, va, vb, vm
va <= vb        vmsle{u}.vv vd, va, vb, vm
va > vb         vmslt{u}.vv vd, vb, va, vm    vmsgt{u}.vv vd, va, vb, vm
va >= vb        vmsle{u}.vv vd, vb, va, vm    vmsge{u}.vv vd, va, vb, vm

va < x          vmslt{u}.vx vd, va, x, vm
va <= x         vmsle{u}.vx vd, va, x, vm
va > x          vmsgt{u}.vx vd, va, x, vm
va >= x         see below

va < i          vmsle{u}.vi vd, va, i-1, vm    vmslt{u}.vi vd, va, i, vm
va <= i         vmsle{u}.vi vd, va, i, vm
va > i          vmsgt{u}.vi vd, va, i, vm
va >= i         vmsgt{u}.vi vd, va, i-1, vm    vmsge{u}.vi vd, va, i, vm

va, vb vector register groups
x      scalar integer register
i      immediate
```

The immediate forms of `vmslt{u}.vi` *are not provided as the immediate value can be decreased by 1 and the* `vmsle{u}.vi` *variants used instead. The* `vmsle.vi` *range is -16 to 15, resulting in an effective* `vmslt.vi` *range of -15 to 16. The* `vmsleu.vi` *range is 0 to 15 giving an effective* `vmsltu.vi` *range of 1 to 16 (Note,* `vmsltu.vi` *with immediate 0 is not useful as it is always false).*

Because the 5-bit vector immediates are always sign-extended, when the high bit of the `simm5` *immediate is set,* `vmsleu.vi` *also supports unsigned immediate values in the range* $2^{SEW}-16$ *to* $2^{SEW}-1$*, allowing corresponding* `vmsltu.vi` *compares against unsigned immediates in the range* $2^{SEW}-15$ *to* 2^{SEW}*. Note that* `vmsltu.vi` *with immediate* 2^{SEW} *is not useful as it is always true.*

Similarly, `vmsge{u}.vi` is not provided and the compare is implemented using `vmsgt{u}.vi` with the immediate decremented by one. The resulting effective `vmsge.vi` range is -15 to 16, and the resulting effective `vmsgeu.vi` range is 1 to 16 (Note, `vmsgeu.vi` with immediate 0 is not useful as it is always true).

The `vmsgt` *forms for register scalar and immediates are provided to allow a single compare instruction to provide the correct polarity of mask value without using additional mask logical instructions.*

To reduce encoding space, the `vmsge{u}.vx` form is not directly provided, and so the `va ≥ x` case requires special treatment.

The `vmsge{u}.vx` *could potentially be encoded in a non-orthogonal way under the unused OPIVI variant of* `vmslt{u}`*. These would be the only instructions in OPIVI that use a scalar* `x` *register however. Alternatively, a further two funct6 encodings could be used, but these would have a different operand format (writes to mask register) than others in the same group of 8 funct6 encodings. The current PoR is to omit these instructions and to synthesize where needed as described below.*

The `vmsge{u}.vx` operation can be synthesized by reducing the value of `x` by 1 and using the `vmsgt{u}.vx` instruction, when it is known that this will not underflow the representation in `x`.

```
Sequences to synthesize vmsge{u}.vx instruction

va >= x,  x > minimum

   addi t0, x, -1; vmsgt{u}.vx vd, va, t0, vm
```

The above sequence will usually be the most efficient implementation, but assembler pseudoinstructions can be provided for cases where the range of `x` is unknown.

```
unmasked va >= x

  pseudoinstruction: vmsge{u}.vx vd, va, x
  expansion: vmslt{u}.vx vd, va, x; vmnand.mm vd, vd, vd

masked va >= x, vd != v0

  pseudoinstruction: vmsge{u}.vx vd, va, x, v0.t
  expansion: vmslt{u}.vx vd, va, x, v0.t; vmxor.mm vd, vd, v0

masked va >= x, vd == v0

  pseudoinstruction: vmsge{u}.vx vd, va, x, v0.t, vt
  expansion: vmslt{u}.vx vt, va, x;  vmandn.mm vd, vd, vt

masked va >= x, any vd

  pseudoinstruction: vmsge{u}.vx vd, va, x, v0.t, vt
  expansion: vmslt{u}.vx vt, va, x;  vmandn.mm vt, v0, vt;  vmandn.mm vd, vd,
v0;  vmor.mm vd, vt, vd

  The vt argument to the pseudoinstruction must name a temporary vector
register that is
  not same as vd and which will be clobbered by the pseudoinstruction
```

Compares effectively AND in the mask under a mask-undisturbed policy if the destination register is `v0`, e.g.,

```
# (a < b) && (b < c) in two instructions when mask-undisturbed
vmslt.vv    v0, va, vb        # All body elements written
vmslt.vv    v0, vb, vc, v0.t  # Only update at set mask
```

Compares write mask registers, and so always operate under a tail-agnostic policy.

31.11.9. Vector Integer Min/Max Instructions

Signed and unsigned integer minimum and maximum instructions are supported.

```
# Unsigned minimum
vminu.vv vd, vs2, vs1, vm   # Vector-vector
vminu.vx vd, vs2, rs1, vm   # vector-scalar

# Signed minimum
vmin.vv vd, vs2, vs1, vm   # Vector-vector
vmin.vx vd, vs2, rs1, vm   # vector-scalar

# Unsigned maximum
vmaxu.vv vd, vs2, vs1, vm   # Vector-vector
vmaxu.vx vd, vs2, rs1, vm   # vector-scalar

# Signed maximum
vmax.vv vd, vs2, vs1, vm   # Vector-vector
vmax.vx vd, vs2, rs1, vm   # vector-scalar
```

31.11.10. Vector Single-Width Integer Multiply Instructions

The single-width multiply instructions perform a SEW-bit*SEW-bit multiply to generate a 2*SEW-bit product, then return one half of the product in the SEW-bit-wide destination. The **`mul`** versions write the low word of the product to the destination register, while the **`mulh`** versions write the high word of the product to the destination register.

```
# Signed multiply, returning low bits of product
vmul.vv vd, vs2, vs1, vm   # Vector-vector
vmul.vx vd, vs2, rs1, vm   # vector-scalar

# Signed multiply, returning high bits of product
vmulh.vv vd, vs2, vs1, vm   # Vector-vector
vmulh.vx vd, vs2, rs1, vm   # vector-scalar

# Unsigned multiply, returning high bits of product
vmulhu.vv vd, vs2, vs1, vm   # Vector-vector
vmulhu.vx vd, vs2, rs1, vm   # vector-scalar

# Signed(vs2)-Unsigned multiply, returning high bits of product
vmulhsu.vv vd, vs2, vs1, vm   # Vector-vector
vmulhsu.vx vd, vs2, rs1, vm   # vector-scalar
```

There is no `vmulhus.vx` *opcode to return high half of unsigned-vector * signed-scalar product. The scalar can be splatted to a vector, then a* `vmulhsu.vv` *used.*

The current `vmulh*` *opcodes perform simple fractional multiplies, but with no option to scale, round, and/or saturate the result. A possible future extension can consider variants of* `vmulh`*,*

`vmulhu`, `vmulhsu` *that use the* `vxrm` *rounding mode when discarding low half of product. There is no possibility of overflow in these cases.*

31.11.11. Vector Integer Divide Instructions

The divide and remainder instructions are equivalent to the RISC-V standard scalar integer multiply/divides, with the same results for extreme inputs.

```
# Unsigned divide.
vdivu.vv vd, vs2, vs1, vm   # Vector-vector
vdivu.vx vd, vs2, rs1, vm   # vector-scalar

# Signed divide
vdiv.vv vd, vs2, vs1, vm   # Vector-vector
vdiv.vx vd, vs2, rs1, vm   # vector-scalar

# Unsigned remainder
vremu.vv vd, vs2, vs1, vm   # Vector-vector
vremu.vx vd, vs2, rs1, vm   # vector-scalar

# Signed remainder
vrem.vv vd, vs2, vs1, vm   # Vector-vector
vrem.vx vd, vs2, rs1, vm   # vector-scalar
```

The decision to include integer divide and remainder was contentious. The argument in favor is that without a standard instruction, software would have to pick some algorithm to perform the operation, which would likely perform poorly on some microarchitectures versus others.

There is no instruction to perform a "scalar divide by vector" operation.

31.11.12. Vector Widening Integer Multiply Instructions

The widening integer multiply instructions return the full 2*SEW-bit product from an SEW-bit*SEW-bit multiply.

```
# Widening signed-integer multiply
vwmul.vv  vd, vs2, vs1, vm # vector-vector
vwmul.vx  vd, vs2, rs1, vm # vector-scalar

# Widening unsigned-integer multiply
vwmulu.vv vd, vs2, vs1, vm # vector-vector
vwmulu.vx vd, vs2, rs1, vm # vector-scalar

# Widening signed(vs2)-unsigned integer multiply
vwmulsu.vv vd, vs2, vs1, vm # vector-vector
vwmulsu.vx vd, vs2, rs1, vm # vector-scalar
```

31.11.13. Vector Single-Width Integer Multiply-Add Instructions

The integer multiply-add instructions are destructive and are provided in two forms, one that overwrites the addend or minuend (`vmacc`, `vnmsac`) and one that overwrites the first multiplicand (`vmadd`, `vnmsub`).

The low half of the product is added or subtracted from the third operand.

`sac` *is intended to be read as "subtract from accumulator". The opcode is* `vnmsac` *to match the (unfortunately counterintuitive) floating-point* `fnmsub` *instruction definition. Similarly for the* `vnmsub` *opcode.*

```
# Integer multiply-add, overwrite addend
vmacc.vv vd, vs1, vs2, vm     # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vmacc.vx vd, rs1, vs2, vm     # vd[i] = (x[rs1] * vs2[i]) + vd[i]

# Integer multiply-sub, overwrite minuend
vnmsac.vv vd, vs1, vs2, vm    # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vnmsac.vx vd, rs1, vs2, vm    # vd[i] = -(x[rs1] * vs2[i]) + vd[i]

# Integer multiply-add, overwrite multiplicand
vmadd.vv vd, vs1, vs2, vm     # vd[i] = (vs1[i] * vd[i]) + vs2[i]
vmadd.vx vd, rs1, vs2, vm     # vd[i] = (x[rs1] * vd[i]) + vs2[i]

# Integer multiply-sub, overwrite multiplicand
vnmsub.vv vd, vs1, vs2, vm    # vd[i] = -(vs1[i] * vd[i]) + vs2[i]
vnmsub.vx vd, rs1, vs2, vm    # vd[i] = -(x[rs1] * vd[i]) + vs2[i]
```

31.11.14. Vector Widening Integer Multiply-Add Instructions

The widening integer multiply-add instructions add the full 2*SEW-bit product from a SEW-bit*SEW-bit multiply to a 2*SEW-bit value and produce a 2*SEW-bit result. All combinations of signed and unsigned multiply operands are supported.

```
# Widening unsigned-integer multiply-add, overwrite addend
vwmaccu.vv vd, vs1, vs2, vm   # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vwmaccu.vx vd, rs1, vs2, vm   # vd[i] = (x[rs1] * vs2[i]) + vd[i]

# Widening signed-integer multiply-add, overwrite addend
vwmacc.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vwmacc.vx vd, rs1, vs2, vm    # vd[i] = (x[rs1] * vs2[i]) + vd[i]

# Widening signed-unsigned-integer multiply-add, overwrite addend
vwmaccsu.vv vd, vs1, vs2, vm  # vd[i] = (signed(vs1[i]) * unsigned(vs2[i])) +
vd[i]
vwmaccsu.vx vd, rs1, vs2, vm  # vd[i] = (signed(x[rs1]) * unsigned(vs2[i])) +
vd[i]

# Widening unsigned-signed-integer multiply-add, overwrite addend
vwmaccus.vx vd, rs1, vs2, vm  # vd[i] = (unsigned(x[rs1]) * signed(vs2[i])) +
```

```
vd[i]
```

31.11.15. Vector Integer Merge Instructions

The vector integer merge instructions combine two source operands based on a mask. Unlike regular arithmetic instructions, the merge operates on all body elements (i.e., the set of elements from `vstart` up to the current vector length in `vl`).

The `vmerge` instructions are encoded as masked instructions (`vm=0`). The instructions combine two sources as follows. At elements where the mask value is zero, the first operand is copied to the destination element, otherwise the second operand is copied to the destination element. The first operand is always a vector register group specified by `vs2`. The second operand is a vector register group specified by `vs1` or a scalar `x` register specified by `rs1` or a 5-bit sign-extended immediate.

```
vmerge.vvm vd, vs2, vs1, v0  # vd[i] = v0.mask[i] ? vs1[i] : vs2[i]
vmerge.vxm vd, vs2, rs1, v0  # vd[i] = v0.mask[i] ? x[rs1] : vs2[i]
vmerge.vim vd, vs2, imm, v0  # vd[i] = v0.mask[i] ? imm    : vs2[i]
```

31.11.16. Vector Integer Move Instructions

The vector integer move instructions copy a source operand to a vector register group. The `vmv.v.v` variant copies a vector register group, whereas the `vmv.v.x` and `vmv.v.i` variants *splat* a scalar register or immediate to all active elements of the destination vector register group. These instructions are encoded as unmasked instructions (`vm=1`). The first operand specifier (`vs2`) must contain `v0`, and any other vector register number in `vs2` is *reserved*.

```
vmv.v.v vd, vs1 # vd[i] = vs1[i]
vmv.v.x vd, rs1 # vd[i] = x[rs1]
vmv.v.i vd, imm # vd[i] = imm
```

Mask values can be widened into SEW-width elements using a sequence `vmv.v.i vd, 0; vmerge.vim vd, vd, 1, v0`.

The vector integer move instructions share the encoding with the vector merge instructions, but with `vm=1` *and* `vs2=v0`.

The form `vmv.v.v vd, vd`, which leaves body elements unchanged, can be used to indicate that the register will next be used with an EEW equal to SEW.

Implementations that internally reorganize data according to EEW can shuffle the internal representation according to SEW. Implementations that do not internally reorganize data can dynamically elide this instruction (aside from resetting `vstart` *to 0).*

The `vmv.v.v vd, vd` *instruction is not a RISC-V HINT as a tail-agnostic setting may cause an architectural state change on some implementations.*

31.12. Vector Fixed-Point Arithmetic Instructions

The preceding set of integer arithmetic instructions is extended to support fixed-point arithmetic.

A fixed-point number is a two's-complement signed or unsigned integer interpreted as the numerator in a fraction with an implicit denominator. The fixed-point instructions are intended to be applied to the numerators; it is the responsibility of software to manage the denominators. An N-bit element can hold two's-complement signed integers in the range $-2^{N-1}...+2^{N-1}-1$, and unsigned integers in the range $0 ... +2^{N}-1$. The fixed-point instructions help preserve precision in narrow operands by supporting scaling and rounding, and can handle overflow by saturating results into the destination format range.

The widening integer operations described above can also be used to avoid overflow.

31.12.1. Vector Single-Width Saturating Add and Subtract

Saturating forms of integer add and subtract are provided, for both signed and unsigned integers. If the result would overflow the destination, the result is replaced with the closest representable value, and the `vxsat` bit is set.

```
# Saturating adds of unsigned integers.
vsaddu.vv vd, vs2, vs1, vm   # Vector-vector
vsaddu.vx vd, vs2, rs1, vm   # vector-scalar
vsaddu.vi vd, vs2, imm, vm   # vector-immediate

# Saturating adds of signed integers.
vsadd.vv vd, vs2, vs1, vm   # Vector-vector
vsadd.vx vd, vs2, rs1, vm   # vector-scalar
vsadd.vi vd, vs2, imm, vm   # vector-immediate

# Saturating subtract of unsigned integers.
vssubu.vv vd, vs2, vs1, vm   # Vector-vector
vssubu.vx vd, vs2, rs1, vm   # vector-scalar

# Saturating subtract of signed integers.
vssub.vv vd, vs2, vs1, vm   # Vector-vector
vssub.vx vd, vs2, rs1, vm   # vector-scalar
```

31.12.2. Vector Single-Width Averaging Add and Subtract

The averaging add and subtract instructions right shift the result by one bit and round off the result according to the setting in `vxrm`. Computation is performed in infinite precision before rounding and truncating. Both unsigned and signed versions are provided. For `vaaddu` and `vaadd` there can be no overflow in the result. For `vasub` and `vasubu`, overflow is ignored and the result wraps around.

For `vasub`, overflow occurs only when subtracting the smallest number from the largest number under `rnu` or `rne` rounding.

```
# Averaging add
```

```
# Averaging adds of unsigned integers.
vaaddu.vv vd, vs2, vs1, vm   # roundoff_unsigned(vs2[i] + vs1[i], 1)
vaaddu.vx vd, vs2, rs1, vm   # roundoff_unsigned(vs2[i] + x[rs1], 1)

# Averaging adds of signed integers.
vaadd.vv vd, vs2, vs1, vm   # roundoff_signed(vs2[i] + vs1[i], 1)
vaadd.vx vd, vs2, rs1, vm   # roundoff_signed(vs2[i] + x[rs1], 1)

# Averaging subtract

# Averaging subtract of unsigned integers.
vasubu.vv vd, vs2, vs1, vm   # roundoff_unsigned(vs2[i] - vs1[i], 1)
vasubu.vx vd, vs2, rs1, vm   # roundoff_unsigned(vs2[i] - x[rs1], 1)

# Averaging subtract of signed integers.
vasub.vv vd, vs2, vs1, vm   # roundoff_signed(vs2[i] - vs1[i], 1)
vasub.vx vd, vs2, rs1, vm   # roundoff_signed(vs2[i] - x[rs1], 1)
```

31.12.3. Vector Single-Width Fractional Multiply with Rounding and Saturation

The signed fractional multiply instruction produces a 2*SEW product of the two SEW inputs, then shifts the result right by SEW-1 bits, rounding these bits according to `vxrm`, then saturates the result to fit into SEW bits. If the result causes saturation, the `vxsat` bit is set.

```
# Signed saturating and rounding fractional multiply
# See vxrm description for rounding calculation
vsmul.vv vd, vs2, vs1, vm  # vd[i] = clip(roundoff_signed(vs2[i]*vs1[i], SEW-
1))
vsmul.vx vd, vs2, rs1, vm  # vd[i] = clip(roundoff_signed(vs2[i]*x[rs1], SEW-
1))
```

When multiplying two N-bit signed numbers, the largest magnitude is obtained for $-2^{N-1} * -2^{N-1}$ producing a result $+2^{2N-2}$, which has a single (zero) sign bit when held in 2N bits. All other products have two sign bits in 2N bits. To retain greater precision in N result bits, the product is shifted right by one bit less than N, saturating the largest magnitude result but increasing result precision by one bit for all other products.

We do not provide an equivalent fractional multiply where one input is unsigned, as these would retain all upper SEW bits and would not need to saturate. This operation is partly covered by the `vmulhu` and `vmulhsu` instructions, for the case where rounding is simply truncation (`rdn`).

31.12.4. Vector Single-Width Scaling Shift Instructions

These instructions shift the input value right, and round off the shifted out bits according to `vxrm`. The scaling right shifts have both zero-extending (`vssrl`) and sign-extending (`vssra`) forms. The data to be shifted is in the vector register group specified by `vs2` and the shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. Only the low lg2(SEW)

bits of the shift-amount value are used to control the shift amount.

```
# Scaling shift right logical
vssrl.vv vd, vs2, vs1, vm   # vd[i] = roundoff_unsigned(vs2[i], vs1[i])
vssrl.vx vd, vs2, rs1, vm   # vd[i] = roundoff_unsigned(vs2[i], x[rs1])
vssrl.vi vd, vs2, uimm, vm  # vd[i] = roundoff_unsigned(vs2[i], uimm)

# Scaling shift right arithmetic
vssra.vv vd, vs2, vs1, vm   # vd[i] = roundoff_signed(vs2[i],vs1[i])
vssra.vx vd, vs2, rs1, vm   # vd[i] = roundoff_signed(vs2[i], x[rs1])
vssra.vi vd, vs2, uimm, vm  # vd[i] = roundoff_signed(vs2[i], uimm)
```

31.12.5. Vector Narrowing Fixed-Point Clip Instructions

The `vnclip` instructions are used to pack a fixed-point value into a narrower destination. The instructions support rounding, scaling, and saturation into the final destination format. The source data is in the vector register group specified by `vs2`. The scaling shift amount value can come from a vector register group `vs1`, a scalar integer register `rs1`, or a zero-extended 5-bit immediate. The low lg2(2*SEW) bits of the vector or scalar shift-amount value (e.g., the low 6 bits for a SEW=64-bit to SEW=32-bit narrowing operation) are used to control the right shift amount, which provides the scaling.

```
# Narrowing unsigned clip
#                                SEW                              2*SEW   SEW
vnclipu.wv vd, vs2, vs1, vm  # vd[i] = clip(roundoff_unsigned(vs2[i], vs1[i]))
vnclipu.wx vd, vs2, rs1, vm  # vd[i] = clip(roundoff_unsigned(vs2[i], x[rs1]))
vnclipu.wi vd, vs2, uimm, vm # vd[i] = clip(roundoff_unsigned(vs2[i], uimm))

# Narrowing signed clip
vnclip.wv vd, vs2, vs1, vm   # vd[i] = clip(roundoff_signed(vs2[i], vs1[i]))
vnclip.wx vd, vs2, rs1, vm   # vd[i] = clip(roundoff_signed(vs2[i], x[rs1]))
vnclip.wi vd, vs2, uimm, vm  # vd[i] = clip(roundoff_signed(vs2[i], uimm))
```

For `vnclipu`/`vnclip`, the rounding mode is specified in the `vxrm` CSR. Rounding occurs around the least-significant bit of the destination and before saturation.

For `vnclipu`, the shifted rounded source value is treated as an unsigned integer and saturates if the result would overflow the destination viewed as an unsigned integer.

There is no single instruction that can saturate a signed value into an unsigned destination. A sequence of two vector instructions that first removes negative numbers by performing a max against 0 using `vmax` *then clips the resulting unsigned value into the destination using* `vnclipu` *can be used if setting* `vxsat` *value for negative numbers is not required. A* `vsetvli` *is required between these two instructions to change SEW.*

For `vnclip`, the shifted rounded source value is treated as a signed integer and saturates if the result would overflow the destination viewed as a signed integer.

If any destination element is saturated, the `vxsat` bit is set in the `vxsat` register.

31.13. Vector Floating-Point Instructions

The standard vector floating-point instructions treat elements as IEEE-754/2008-compatible values. If the EEW of a vector floating-point operand does not correspond to a supported IEEE floating-point type, the instruction encoding is reserved.

Whether floating-point is supported, and for which element widths, is determined by the specific vector extension. The current set of extensions include support for 32-bit and 64-bit floating-point values. When 16-bit and 128-bit element widths are added, they will be also be treated as IEEE-754/2008-compatible values. Other floating-point formats may be supported in future extensions.

Vector floating-point instructions require the presence of base scalar floating-point extensions corresponding to the supported vector floating-point element widths.

In particular, future vector extensions supporting 16-bit half-precision floating-point values will also require some scalar half-precision floating-point support.

If the floating-point unit status field `mstatus.FS` is `Off` then any attempt to execute a vector floating-point instruction will raise an illegal-instruction exception. Any vector floating-point instruction that modifies any floating-point extension state (i.e., floating-point CSRs or `f` registers) must set `mstatus.FS` to `Dirty`.

If the hypervisor extension is implemented and V=1, the `vsstatus.FS` field is additionally in effect for vector floating-point instructions. If `vsstatus.FS` or `mstatus.FS` is `Off` then any attempt to execute a vector floating-point instruction will raise an illegal-instruction exception. Any vector floating-point instruction that modifies any floating-point extension state (i.e., floating-point CSRs or `f` registers) must set both `mstatus.FS` and `vsstatus.FS` to `Dirty`.

The vector floating-point instructions have the same behavior as the scalar floating-point instructions with regard to NaNs.

Scalar values for floating-point vector-scalar operations are sourced as described in Section 31.10.1.

31.13.1. Vector Floating-Point Exception Flags

A vector floating-point exception at any active floating-point element sets the standard FP exception flags in the `fflags` register. Inactive elements do not set FP exception flags.

31.13.2. Vector Single-Width Floating-Point Add/Subtract Instructions

```
# Floating-point add
vfadd.vv vd, vs2, vs1, vm   # Vector-vector
vfadd.vf vd, vs2, rs1, vm   # vector-scalar

# Floating-point subtract
vfsub.vv vd, vs2, vs1, vm   # Vector-vector
vfsub.vf vd, vs2, rs1, vm   # Vector-scalar vd[i] = vs2[i] - f[rs1]
vfrsub.vf vd, vs2, rs1, vm  # Scalar-vector vd[i] = f[rs1] - vs2[i]
```

31.13.3. Vector Widening Floating-Point Add/Subtract Instructions

```
# Widening FP add/subtract, 2*SEW = SEW +/- SEW
vfwadd.vv vd, vs2, vs1, vm  # vector-vector
vfwadd.vf vd, vs2, rs1, vm  # vector-scalar
vfwsub.vv vd, vs2, vs1, vm  # vector-vector
vfwsub.vf vd, vs2, rs1, vm  # vector-scalar

# Widening FP add/subtract, 2*SEW = 2*SEW +/- SEW
vfwadd.wv  vd, vs2, vs1, vm  # vector-vector
vfwadd.wf  vd, vs2, rs1, vm  # vector-scalar
vfwsub.wv  vd, vs2, vs1, vm  # vector-vector
vfwsub.wf  vd, vs2, rs1, vm  # vector-scalar
```

31.13.4. Vector Single-Width Floating-Point Multiply/Divide Instructions

```
# Floating-point multiply
vfmul.vv vd, vs2, vs1, vm   # Vector-vector
vfmul.vf vd, vs2, rs1, vm   # vector-scalar

# Floating-point divide
vfdiv.vv vd, vs2, vs1, vm   # Vector-vector
vfdiv.vf vd, vs2, rs1, vm   # vector-scalar

# Reverse floating-point divide vector = scalar / vector
vfrdiv.vf vd, vs2, rs1, vm  # scalar-vector, vd[i] = f[rs1]/vs2[i]
```

31.13.5. Vector Widening Floating-Point Multiply

```
# Widening floating-point multiply
vfwmul.vv    vd, vs2, vs1, vm # vector-vector
vfwmul.vf    vd, vs2, rs1, vm # vector-scalar
```

31.13.6. Vector Single-Width Floating-Point Fused Multiply-Add Instructions

All four varieties of fused multiply-add are provided, and in two destructive forms that overwrite one of the operands, either the addend or the first multiplicand.

```
# FP multiply-accumulate, overwrites addend
vfmacc.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vfmacc.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vs2[i]) + vd[i]

# FP negate-(multiply-accumulate), overwrites subtrahend
vfnmacc.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
```

```
vfnmacc.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

# FP multiply-subtract-accumulator, overwrites subtrahend
vfmsac.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) - vd[i]
vfmsac.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vs2[i]) - vd[i]

# FP negate-(multiply-subtract-accumulator), overwrites minuend
vfnmsac.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vfnmsac.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) + vd[i]

# FP multiply-add, overwrites multiplicand
vfmadd.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vd[i]) + vs2[i]
vfmadd.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vd[i]) + vs2[i]

# FP negate-(multiply-add), overwrites multiplicand
vfnmadd.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vd[i]) - vs2[i]
vfnmadd.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vd[i]) - vs2[i]

# FP multiply-sub, overwrites multiplicand
vfmsub.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vd[i]) - vs2[i]
vfmsub.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vd[i]) - vs2[i]

# FP negate-(multiply-sub), overwrites multiplicand
vfnmsub.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vd[i]) + vs2[i]
vfnmsub.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vd[i]) + vs2[i]
```

While we considered using the two unused rounding modes in the scalar FP FMA encoding to provide a few non-destructive FMAs, these would complicate microarchitectures by being the only maskable operation with three inputs and separate output.

31.13.7. Vector Widening Floating-Point Fused Multiply-Add Instructions

The widening floating-point fused multiply-add instructions all overwrite the wide addend with the result. The multiplier inputs are all SEW wide, while the addend and destination is 2*SEW bits wide.

```
# FP widening multiply-accumulate, overwrites addend
vfwmacc.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vfwmacc.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vs2[i]) + vd[i]

# FP widening negate-(multiply-accumulate), overwrites addend
vfwnmacc.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfwnmacc.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

# FP widening multiply-subtract-accumulator, overwrites addend
vfwmsac.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) - vd[i]
vfwmsac.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vs2[i]) - vd[i]

# FP widening negate-(multiply-subtract-accumulator), overwrites addend
vfwnmsac.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
```

```
vfwnmsac.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) + vd[i]
```

31.13.8. Vector Floating-Point Square-Root Instruction

This is a unary vector-vector instruction.

```
# Floating-point square root
vfsqrt.v vd, vs2, vm   # Vector-vector square root
```

31.13.9. Vector Floating-Point Reciprocal Square-Root Estimate Instruction

```
# Floating-point reciprocal square-root estimate to 7 bits.
vfrsqrt7.v vd, vs2, vm
```

This is a unary vector-vector instruction that returns an estimate of 1/sqrt(x) accurate to 7 bits.

An earlier draft version had used the assembler name `vfrsqrte7` *but this was deemed to cause confusion with the* `e`*x notation for element width. The earlier name can be retained as alias in tool chains for backward compatibility.*

The following table describes the instruction's behavior for all classes of floating-point inputs:

Input	Output	Exceptions raised
$-\infty \le x < -0.0$	canonical NaN	NV
-0.0	$-\infty$	DZ
+0.0	$+\infty$	DZ
$+0.0 < x < +\infty$	*estimate of 1/sqrt(x)*	
$+\infty$	+0.0	
qNaN	canonical NaN	
sNaN	canonical NaN	NV

All positive normal and subnormal inputs produce normal outputs.

The output value is independent of the dynamic rounding mode.

For the non-exceptional cases, the low bit of the exponent and the six high bits of significand (after the leading one) are concatenated and used to address the following table. The output of the table becomes the seven high bits of the result significand (after the leading one); the remainder of the result significand is zero. Subnormal inputs are normalized and the exponent adjusted appropriately before the lookup. The output exponent is chosen to make the result approximate the reciprocal of the square root of the argument.

More precisely, the result is computed as follows. Let the normalized input exponent be equal to the input exponent if the input is normal, or 0 minus the number of leading zeros in the significand otherwise. If the input is subnormal, the normalized input significand is given by shifting the input significand left by 1 minus the normalized input exponent, discarding the leading 1 bit. The output exponent equals floor((3*B -

1 - the normalized input exponent) / 2), where B is the exponent bias. The output sign equals the input sign.

The following table gives the seven MSBs of the output significand as a function of the LSB of the normalized input exponent and the six MSBs of the normalized input significand; the other bits of the output significand are zero.

Table 57. vfrsqrt7.v common-case lookup table contents

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
0	0	52
0	1	51
0	2	50
0	3	48
0	4	47
0	5	46
0	6	44
0	7	43
0	8	42
0	9	41
0	10	40
0	11	39
0	12	38
0	13	36
0	14	35
0	15	34
0	16	33
0	17	32
0	18	31
0	19	30
0	20	30
0	21	29
0	22	28
0	23	27
0	24	26
0	25	25
0	26	24
0	27	23
0	28	23
0	29	22
0	30	21
0	31	20
0	32	19
0	33	19

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
0	34	18
0	35	17
0	36	16
0	37	16
0	38	15
0	39	14
0	40	14
0	41	13
0	42	12
0	43	12
0	44	11
0	45	10
0	46	10
0	47	9
0	48	9
0	49	8
0	50	7
0	51	7
0	52	6
0	53	6
0	54	5
0	55	4
0	56	4
0	57	3
0	58	3
0	59	2
0	60	2
0	61	1
0	62	1
0	63	0
1	0	127
1	1	125
1	2	123
1	3	121
1	4	119
1	5	118
1	6	116
1	7	114
1	8	113

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
1	9	111
1	10	109
1	11	108
1	12	106
1	13	105
1	14	103
1	15	102
1	16	100
1	17	99
1	18	97
1	19	96
1	20	95
1	21	93
1	22	92
1	23	91
1	24	90
1	25	88
1	26	87
1	27	86
1	28	85
1	29	84
1	30	83
1	31	82
1	32	80
1	33	79
1	34	78
1	35	77
1	36	76
1	37	75
1	38	74
1	39	73
1	40	72
1	41	71
1	42	70
1	43	70
1	44	69
1	45	68
1	46	67
1	47	66

exp[0]	sig[MSB -: 6]	sig_out[MSB -: 7]
1	48	65
1	49	64
1	50	63
1	51	63
1	52	62
1	53	61
1	54	60
1	55	59
1	56	59
1	57	58
1	58	57
1	59	56
1	60	56
1	61	55
1	62	54
1	63	53

For example, when SEW=32, vfrsqrt7(0x00718abc (≈ 1.043e-38)) = 0x5f080000 (≈ 9.800e18), and vfrsqrt7(0x7f765432 (≈ 3.274e38)) = 0x1f820000 (≈ 5.506e-20).

The 7 bit accuracy was chosen as it requires 0,1,2,3 Newton-Raphson iterations to converge to close to bfloat16, FP16, FP32, FP64 accuracy respectively. Future instructions can be defined with greater estimate accuracy.

31.13.10. Vector Floating-Point Reciprocal Estimate Instruction

```
# Floating-point reciprocal estimate to 7 bits.
vfrec7.v vd, vs2, vm
```

An earlier draft version had used the assembler name `vfrece7` but this was deemed to cause confusion with `e`x notation for element width. The earlier name can be retained as alias in tool chains for backward compatibility.

This is a unary vector-vector instruction that returns an estimate of 1/x accurate to 7 bits.

The following table describes the instruction's behavior for all classes of floating-point inputs, where *B* is the exponent bias:

Input (x)	Rounding Mode	Output ($y \approx 1/x$)	Exceptions raised
$-\infty$	*any*	-0.0	
$-2^{B+1} < x \le -2^{B}$ (normal)	*any*	$-2^{-(B+1)} \ge y > -2^{-B}$ (subnormal, sig=01...)	
$-2^{B} < x \le -2^{B-1}$ (normal)	*any*	$-2^{-B} \ge y > -2^{-B+1}$ (subnormal, sig=1...)	
$-2^{B-1} < x \le -2^{-B+1}$ (normal)	*any*	$-2^{-B+1} \ge y > -2^{B-1}$ (normal)	

Input (x)	Rounding Mode	Output ($y \approx 1/x$)	Exceptions raised
$-2^{-B+1} < x \le -2^{-B}$ (subnormal, sig=1...)	*any*	$-2^{B-1} \ge y > -2^{B}$ (normal)	
$-2^{-B} < x \le -2^{-(B+1)}$ (subnormal, sig=01...)	*any*	$-2^{B} \ge y > -2^{B+1}$ (normal)	
$-2^{-(B+1)} < x < -0.0$ (subnormal, sig=00...)	RUP, RTZ	greatest-mag. negative finite value	NX, OF
$-2^{-(B+1)} < x < -0.0$ (subnormal, sig=00...)	RDN, RNE, RMM	$-\infty$	NX, OF
-0.0	*any*	$-\infty$	DZ
+0.0	*any*	$+\infty$	DZ
$+0.0 < x < 2^{-(B+1)}$ (subnormal, sig=00...)	RUP, RNE, RMM	$+\infty$	NX, OF
$+0.0 < x < 2^{-(B+1)}$ (subnormal, sig=00...)	RDN, RTZ	greatest finite value	NX, OF
$2^{-(B+1)} \le x < 2^{-B}$ (subnormal, sig=01...)	*any*	$2^{B+1} > y \ge 2^{B}$ (normal)	
$2^{-B} \le x < 2^{-B+1}$ (subnormal, sig=1...)	*any*	$2^{B} > y \ge 2^{B-1}$ (normal)	
$2^{-B+1} \le x < 2^{B-1}$ (normal)	*any*	$2^{B-1} > y \ge 2^{-B+1}$ (normal)	
$2^{B-1} \le x < 2^{B}$ (normal)	*any*	$2^{-B+1} > y \ge 2^{-B}$ (subnormal, sig=1...)	
$2^{B} \le x < 2^{B+1}$ (normal)	*any*	$2^{-B} > y \ge 2^{-(B+1)}$ (subnormal, sig=01...)	
$+\infty$	*any*	+0.0	
qNaN	*any*	canonical NaN	
sNaN	*any*	canonical NaN	NV

Subnormal inputs with magnitude at least $2^{-(B+1)}$ produce normal outputs; other subnormal inputs produce infinite outputs. Normal inputs with magnitude at least 2^{B-1} produce subnormal outputs; other normal inputs produce normal outputs.

The output value depends on the dynamic rounding mode when the overflow exception is raised.

For the non-exceptional cases, the seven high bits of significand (after the leading one) are used to address the following table. The output of the table becomes the seven high bits of the result significand (after the leading one); the remainder of the result significand is zero. Subnormal inputs are normalized and the exponent adjusted appropriately before the lookup. The output exponent is chosen to make the result approximate the reciprocal of the argument, and subnormal outputs are denormalized accordingly.

More precisely, the result is computed as follows. Let the normalized input exponent be equal to the input exponent if the input is normal, or 0 minus the number of leading zeros in the significand otherwise. The normalized output exponent equals (2*B - 1 - the normalized input exponent). If the normalized output exponent is outside the range [-1, 2*B], the result corresponds to one of the exceptional cases in the table above.

If the input is subnormal, the normalized input significand is given by shifting the input significand left by 1 minus the normalized input exponent, discarding the leading 1 bit. Otherwise, the normalized input significand equals the input significand. The following table gives the seven MSBs of the normalized output significand as a function of the seven MSBs of the normalized input significand; the other bits of the normalized output significand are zero.

Table 58. vfrec7.v common-case lookup table contents

sig[MSB -: 7]	sig_out[MSB -: 7]
0	127
1	125

sig[MSB -: 7]	sig_out[MSB -: 7]
2	123
3	121
4	119
5	117
6	116
7	114
8	112
9	110
10	109
11	107
12	105
13	104
14	102
15	100
16	99
17	97
18	96
19	94
20	93
21	91
22	90
23	88
24	87
25	85
26	84
27	83
28	81
29	80
30	79
31	77
32	76
33	75
34	74
35	72
36	71
37	70
38	69
39	68
40	66

sig[MSB -: 7]	sig_out[MSB -: 7]
41	65
42	64
43	63
44	62
45	61
46	60
47	59
48	58
49	57
50	56
51	55
52	54
53	53
54	52
55	51
56	50
57	49
58	48
59	47
60	46
61	45
62	44
63	43
64	42
65	41
66	40
67	40
68	39
69	38
70	37
71	36
72	35
73	35
74	34
75	33
76	32
77	31
78	31
79	30

sig[MSB -: 7]	sig_out[MSB -: 7]
80	29
81	28
82	28
83	27
84	26
85	25
86	25
87	24
88	23
89	23
90	22
91	21
92	21
93	20
94	19
95	19
96	18
97	17
98	17
99	16
100	15
101	15
102	14
103	14
104	13
105	12
106	12
107	11
108	11
109	10
110	9
111	9
112	8
113	8
114	7
115	7
116	6
117	5
118	5

sig[MSB -: 7]	sig_out[MSB -: 7]
119	4
120	4
121	3
122	3
123	2
124	2
125	1
126	1
127	0

If the normalized output exponent is 0 or -1, the result is subnormal: the output exponent is 0, and the output significand is given by concatenating a 1 bit to the left of the normalized output significand, then shifting that quantity right by 1 minus the normalized output exponent. Otherwise, the output exponent equals the normalized output exponent, and the output significand equals the normalized output significand. The output sign equals the input sign.

For example, when SEW=32, vfrec7(0x00718abc (≈ 1.043e-38)) = 0x7e900000 (≈ 9.570e37), and vfrec7(0x7f765432 (≈ 3.274e38)) = 0x00214000 (≈ 3.053e-39).

The 7 bit accuracy was chosen as it requires 0,1,2,3 Newton-Raphson iterations to converge to close to bfloat16, FP16, FP32, FP64 accuracy respectively. Future instructions can be defined with greater estimate accuracy.

31.13.11. Vector Floating-Point MIN/MAX Instructions

The vector floating-point `vfmin` and `vfmax` instructions have the same behavior as the corresponding scalar floating-point instructions in version 2.2 of the RISC-V F/D/Q extension: they perform the `minimumNumber` or `maximumNumber` operation on active elements.

```
# Floating-point minimum
vfmin.vv vd, vs2, vs1, vm   # Vector-vector
vfmin.vf vd, vs2, rs1, vm   # vector-scalar

# Floating-point maximum
vfmax.vv vd, vs2, vs1, vm   # Vector-vector
vfmax.vf vd, vs2, rs1, vm   # vector-scalar
```

31.13.12. Vector Floating-Point Sign-Injection Instructions

Vector versions of the scalar sign-injection instructions. The result takes all bits except the sign bit from the vector `vs2` operands.

```
vfsgnj.vv vd, vs2, vs1, vm   # Vector-vector
vfsgnj.vf vd, vs2, rs1, vm   # vector-scalar

vfsgnjn.vv vd, vs2, vs1, vm  # Vector-vector
```

```
vfsgnjn.vf vd, vs2, rs1, vm  # vector-scalar

vfsgnjx.vv vd, vs2, vs1, vm  # Vector-vector
vfsgnjx.vf vd, vs2, rs1, vm  # vector-scalar
```

A vector of floating-point values can be negated using a sign-injection instruction with both source operands set to the same vector operand. An assembly pseudoinstruction is provided: `vfneg.v vd,vs` = `vfsgnjn.vv vd,vs,vs`.

The absolute value of a vector of floating-point elements can be calculated using a sign-injection instruction with both source operands set to the same vector operand. An assembly pseudoinstruction is provided: `vfabs.v vd,vs` = `vfsgnjx.vv vd,vs,vs`.

31.13.13. Vector Floating-Point Compare Instructions

These vector FP compare instructions compare two source operands and write the comparison result to a mask register. The destination mask vector is always held in a single vector register, with a layout of elements as described in Section 31.4.5. The destination mask vector register may be the same as the source vector mask register (`v0`). Compares write mask registers, and so always operate under a tail-agnostic policy.

The compare instructions follow the semantics of the scalar floating-point compare instructions. `vmfeq` and `vmfne` raise the invalid operation exception only on signaling NaN inputs. `vmflt`, `vmfle`, `vmfgt`, and `vmfge` raise the invalid operation exception on both signaling and quiet NaN inputs. `vmfne` writes 1 to the destination element when either operand is NaN, whereas the other compares write 0 when either operand is NaN.

```
# Compare equal
vmfeq.vv vd, vs2, vs1, vm  # Vector-vector
vmfeq.vf vd, vs2, rs1, vm  # vector-scalar

# Compare not equal
vmfne.vv vd, vs2, vs1, vm  # Vector-vector
vmfne.vf vd, vs2, rs1, vm  # vector-scalar

# Compare less than
vmflt.vv vd, vs2, vs1, vm  # Vector-vector
vmflt.vf vd, vs2, rs1, vm  # vector-scalar

# Compare less than or equal
vmfle.vv vd, vs2, vs1, vm  # Vector-vector
vmfle.vf vd, vs2, rs1, vm  # vector-scalar

# Compare greater than
vmfgt.vf vd, vs2, rs1, vm  # vector-scalar

# Compare greater than or equal
vmfge.vf vd, vs2, rs1, vm  # vector-scalar
```

```
Comparison      Assembler Mapping             Assembler pseudoinstruction

va < vb         vmflt.vv vd, va, vb, vm
va <= vb        vmfle.vv vd, va, vb, vm
va > vb         vmflt.vv vd, vb, va, vm    vmfgt.vv vd, va, vb, vm
va >= vb        vmfle.vv vd, vb, va, vm    vmfge.vv vd, va, vb, vm

va < f          vmflt.vf vd, va, f, vm
va <= f         vmfle.vf vd, va, f, vm
va > f          vmfgt.vf vd, va, f, vm
va >= f         vmfge.vf vd, va, f, vm

va, vb vector register groups
f      scalar floating-point register
```

Providing all forms is necessary to correctly handle unordered compares for NaNs.

C99 floating-point quiet compares can be implemented by masking the signaling compares when either input is NaN, as follows. When the comparand is a non-NaN constant, the middle two instructions can be omitted.

```
# Example of implementing isgreater()
vmfeq.vv v0, va, va        # Only set where A is not NaN.
vmfeq.vv v1, vb, vb        # Only set where B is not NaN.
vmand.mm v0, v0, v1        # Only set where A and B are ordered,
vmfgt.vv v0, va, vb, v0.t  #  so only set flags on ordered values.
```

In the above sequence, it is tempting to mask the second `vmfeq` *instruction and remove the* `vmand` *instruction, but this more efficient sequence incorrectly fails to raise the invalid exception when an element of* `va` *contains a quiet NaN and the corresponding element in* `vb` *contains a signaling NaN.*

31.13.14. Vector Floating-Point Classify Instruction

This is a unary vector-vector instruction that operates in the same way as the scalar classify instruction.

```
vfclass.v vd, vs2, vm   # Vector-vector
```

The 10-bit mask produced by this instruction is placed in the least-significant bits of the result elements. The upper (SEW-10) bits of the result are filled with zeros. The instruction is only defined for SEW=16b and above, so the result will always fit in the destination elements.

31.13.15. Vector Floating-Point Merge Instruction

A vector-scalar floating-point merge instruction is provided, which operates on all body elements from `vstart` up to the current vector length in `vl` regardless of mask value.

The `vfmerge.vfm` instruction is encoded as a masked instruction (`vm=0`). At elements where the mask value is zero, the first vector operand is copied to the destination element, otherwise a scalar floating-point register value is copied to the destination element.

```
vfmerge.vfm vd, vs2, rs1, v0  # vd[i] = v0.mask[i] ? f[rs1] : vs2[i]
```

31.13.16. Vector Floating-Point Move Instruction

The vector floating-point move instruction *splats* a floating-point scalar operand to a vector register group. The instruction copies a scalar `f` register value to all active elements of a vector register group. This instruction is encoded as an unmasked instruction (`vm=1`). The instruction must have the `vs2` field set to `v0`, with all other values for `vs2` reserved.

```
vfmv.v.f vd, rs1  # vd[i] = f[rs1]
```

The `vfmv.v.f` instruction shares the encoding with the `vfmerge.vfm` instruction, but with `vm=1` and `vs2=v0`.

31.13.17. Single-Width Floating-Point/Integer Type-Convert Instructions

Conversion operations are provided to convert to and from floating-point values and unsigned and signed integers, where both source and destination are SEW wide.

```
vfcvt.xu.f.v vd, vs2, vm       # Convert float to unsigned integer.
vfcvt.x.f.v  vd, vs2, vm       # Convert float to signed integer.

vfcvt.rtz.xu.f.v vd, vs2, vm   # Convert float to unsigned integer, truncating.
vfcvt.rtz.x.f.v  vd, vs2, vm   # Convert float to signed integer, truncating.

vfcvt.f.xu.v vd, vs2, vm       # Convert unsigned integer to float.
vfcvt.f.x.v  vd, vs2, vm       # Convert signed integer to float.
```

The conversions follow the same rules on exceptional conditions as the scalar conversion instructions. The conversions use the dynamic rounding mode in `frm`, except for the `rtz` variants, which round towards zero.

The `rtz` variants are provided to accelerate truncating conversions from floating-point to integer, as is common in languages like C and Java.

31.13.18. Widening Floating-Point/Integer Type-Convert Instructions

A set of conversion instructions is provided to convert between narrower integer and floating-point datatypes to a type of twice the width.

```
vfwcvt.xu.f.v vd, vs2, vm       # Convert float to double-width unsigned
integer.
vfwcvt.x.f.v  vd, vs2, vm       # Convert float to double-width signed integer.
```

```
vfwcvt.rtz.xu.f.v vd, vs2, vm   # Convert float to double-width unsigned
integer, truncating.
vfwcvt.rtz.x.f.v  vd, vs2, vm   # Convert float to double-width signed integer,
truncating.

vfwcvt.f.xu.v vd, vs2, vm       # Convert unsigned integer to double-width
float.
vfwcvt.f.x.v  vd, vs2, vm       # Convert signed integer to double-width float.

vfwcvt.f.f.v vd, vs2, vm        # Convert single-width float to double-width
float.
```

These instructions have the same constraints on vector register overlap as other widening instructions (see Section 31.10.2).

A double-width IEEE floating-point value can always represent a single-width integer exactly.

A double-width IEEE floating-point value can always represent a single-width IEEE floating-point value exactly.

A full set of floating-point widening conversions is not supported as single instructions, but any widening conversion can be implemented as several doubling steps with equivalent results and no additional exception flags raised.

31.13.19. Narrowing Floating-Point/Integer Type-Convert Instructions

A set of conversion instructions is provided to convert wider integer and floating-point datatypes to a type of half the width.

```
vfncvt.xu.f.w vd, vs2, vm       # Convert double-width float to unsigned
integer.
vfncvt.x.f.w  vd, vs2, vm       # Convert double-width float to signed integer.

vfncvt.rtz.xu.f.w vd, vs2, vm   # Convert double-width float to unsigned
integer, truncating.
vfncvt.rtz.x.f.w  vd, vs2, vm   # Convert double-width float to signed integer,
truncating.

vfncvt.f.xu.w vd, vs2, vm       # Convert double-width unsigned integer to
float.
vfncvt.f.x.w  vd, vs2, vm       # Convert double-width signed integer to float.

vfncvt.f.f.w vd, vs2, vm        # Convert double-width float to single-width
float.
vfncvt.rod.f.f.w vd, vs2, vm    # Convert double-width float to single-width
float,
                                #  rounding towards odd.
```

These instructions have the same constraints on vector register overlap as other narrowing instructions (see Section 31.10.3).

A full set of floating-point narrowing conversions is not supported as single instructions. Conversions can be implemented in a sequence of halving steps. Results are equivalently rounded and the same exception flags are raised if all but the last halving step use round-towards-odd (`vfncvt.rod.f.f.w`). Only the final step should use the desired rounding mode.

For `vfncvt.rod.f.f.w`, a finite value that exceeds the range of the destination format is converted to the destination format's largest finite value with the same sign.

31.14. Vector Reduction Operations

Vector reduction operations take a vector register group of elements and a scalar held in element 0 of a vector register, and perform a reduction using some binary operator, to produce a scalar result in element 0 of a vector register. The scalar input and output operands are held in element 0 of a single vector register, not a vector register group, so any vector register can be the scalar source or destination of a vector reduction regardless of LMUL setting.

The destination vector register can overlap the source operands, including the mask register.

Vector reductions read and write the scalar operand and result into element 0 of a vector register instead of a scalar register to avoid a loss of decoupling with the scalar processor, and to support future polymorphic use with future types not supported in the scalar unit.

Inactive elements from the source vector register group are excluded from the reduction, but the scalar operand is always included regardless of the mask values.

The other elements in the destination vector register (0 < index < VLEN/SEW) are considered the tail and are managed with the current tail agnostic/undisturbed policy.

If `vl`=0, no operation is performed and the destination register is not updated.

This choice of behavior for `vl`=0 reduces implementation complexity as it is consistent with other operations on vector register state. For the common case that the source and destination scalar operand are the same vector register, this behavior also produces the expected result. For the uncommon case that the source and destination scalar operand are in different vector registers, this instruction will not copy the source into the destination when `vl`=0. However, it is expected that in most of these cases it will be statically known that `vl` is not zero. In other cases, a check for `vl`=0 will have to be added to ensure that the source scalar is copied to the destination (e.g., by explicitly setting `vl`=1 and performing a register-register copy).

Traps on vector reduction instructions are always reported with a `vstart` of 0. Vector reduction operations raise an illegal-instruction exception if `vstart` is non-zero.

The assembler syntax for a reduction operation is `vredop.vs`, where the `.vs` suffix denotes the first operand is a vector register group and the second operand is a scalar stored in element 0 of a vector register.

31.14.1. Vector Single-Width Integer Reduction Instructions

All operands and results of single-width reduction instructions have the same SEW width. Overflows wrap around on arithmetic sums.

```
# Simple reductions, where [*] denotes all active elements:
vredsum.vs  vd, vs2, vs1, vm   # vd[0] =  sum( vs1[0] , vs2[*] )
vredmaxu.vs vd, vs2, vs1, vm   # vd[0] = maxu( vs1[0] , vs2[*] )
vredmax.vs  vd, vs2, vs1, vm   # vd[0] =  max( vs1[0] , vs2[*] )
vredminu.vs vd, vs2, vs1, vm   # vd[0] = minu( vs1[0] , vs2[*] )
vredmin.vs  vd, vs2, vs1, vm   # vd[0] =  min( vs1[0] , vs2[*] )
vredand.vs  vd, vs2, vs1, vm   # vd[0] =  and( vs1[0] , vs2[*] )
vredor.vs   vd, vs2, vs1, vm   # vd[0] =   or( vs1[0] , vs2[*] )
vredxor.vs  vd, vs2, vs1, vm   # vd[0] =  xor( vs1[0] , vs2[*] )
```

31.14.2. Vector Widening Integer Reduction Instructions

The unsigned `vwredsumu.vs` instruction zero-extends the SEW-wide vector elements before summing them, then adds the 2*SEW-width scalar element, and stores the result in a 2*SEW-width scalar element.

The `vwredsum.vs` instruction sign-extends the SEW-wide vector elements before summing them.

For both `vwredsumu.vs` and `vwredsum.vs`, overflows wrap around.

```
# Unsigned sum reduction into double-width accumulator
vwredsumu.vs vd, vs2, vs1, vm   # 2*SEW = 2*SEW + sum(zero-extend(SEW))

# Signed sum reduction into double-width accumulator
vwredsum.vs  vd, vs2, vs1, vm   # 2*SEW = 2*SEW + sum(sign-extend(SEW))
```

31.14.3. Vector Single-Width Floating-Point Reduction Instructions

```
# Simple reductions.
vfredosum.vs vd, vs2, vs1, vm # Ordered sum
vfredusum.vs vd, vs2, vs1, vm # Unordered sum
vfredmax.vs  vd, vs2, vs1, vm # Maximum value
vfredmin.vs  vd, vs2, vs1, vm # Minimum value
```

Older assembler mnemonic `vfredsum` *is retained as alias for* `vfredusum`.

31.14.3.1. Vector Ordered Single-Width Floating-Point Sum Reduction

The `vfredosum` instruction must sum the floating-point values in element order, starting with the scalar in `vs1[0]`--that is, it performs the computation:

```
vd[0] = (((vs1[0] + vs2[0]) + vs2[1]) + ...) + vs2[vl-1]
```

where each addition operates identically to the scalar floating-point instructions in terms of raising exception flags and generating or propagating special values.

The ordered reduction supports compiler auto-vectorization, while the unordered FP sum allows for faster implementations.

When the operation is masked (`vm=0`), the masked-off elements do not affect the result or the exception flags.

If no elements are active, no additions are performed, so the scalar in `vs1[0]` is simply copied to the destination register, without canonicalizing NaN values and without setting any exception flags. This behavior preserves the handling of NaNs, exceptions, and rounding when auto-vectorizing a scalar summation loop.

31.14.3.2. Vector Unordered Single-Width Floating-Point Sum Reduction

The unordered sum reduction instruction, `vfredusum`, provides an implementation more freedom in performing the reduction.

The implementation must produce a result equivalent to a reduction tree composed of binary operator nodes, with the inputs being elements from the source vector register group (`vs2`) and the source scalar value (`vs1[0]`). Each operator in the tree accepts two inputs and produces one result. Each operator first computes an exact sum as a RISC-V scalar floating-point addition with infinite exponent range and precision, then converts this exact sum to a floating-point format with range and precision each at least as great as the element floating-point format indicated by SEW, rounding using the currently active floating-point dynamic rounding mode and raising exception flags as necessary. A different floating-point range and precision may be chosen for the result of each operator. A node where one input is derived only from elements masked-off or beyond the active vector length may either treat that input as the additive identity of the appropriate EEW or simply copy the other input to its output. The rounded result from the root node in the tree is converted (rounded again, using the dynamic rounding mode) to the standard floating-point format indicated by SEW. An implementation is allowed to add an additional additive identity to the final result.

The additive identity is +0.0 when rounding down (towards -∞) or -0.0 for all other rounding modes.

The reduction tree structure must be deterministic for a given value in `vtype` and `vl`.

As a consequence of this definition, implementations need not propagate NaN payloads through the reduction tree when no elements are active. In particular, if no elements are active and the scalar input is NaN, implementations are permitted to canonicalize the NaN and, if the NaN is signaling, set the invalid exception flag. Implementations are alternatively permitted to pass through the original NaN and set no exception flags, as with `vfredosum`.

The `vfredosum` instruction is a valid implementation of the `vfredusum` instruction.

31.14.3.3. Vector Single-Width Floating-Point Max and Min Reductions

The `vfredmin` and `vfredmax` instructions reduce the scalar argument in `vs1[0]` and active elements in `vs2` using the `minimumNumber` and `maximumNumber` operations, respectively.

Floating-point max and min reductions should return the same final value and raise the same exception flags regardless of operation order.

If no elements are active, the scalar in `vs1[0]` is simply copied to the destination register, without canonicalizing NaN values and without setting any exception flags.

31.14.4. Vector Widening Floating-Point Reduction Instructions

Widening forms of the sum reductions are provided that read and write a double-width reduction result.

```
# Simple reductions.
vfwredosum.vs vd, vs2, vs1, vm # Ordered sum
vfwredusum.vs vd, vs2, vs1, vm # Unordered sum
```

Older assembler mnemonic `vfwredsum` *is retained as alias for* `vfwredusum`.

The reduction of the SEW-width elements is performed as in the single-width reduction case, with the elements in `vs2` promoted to 2*SEW bits before adding to the 2*SEW-bit accumulator.

`vfwredosum.vs` *handles inactive elements and NaN payloads analogously to* `vfredosum.vs`*;* `vfwredusum.vs` *does so analogously to* `vfredusum.vs`.

31.15. Vector Mask Instructions

Several instructions are provided to help operate on mask values held in a vector register.

31.15.1. Vector Mask-Register Logical Instructions

Vector mask-register logical operations operate on mask registers. Each element in a mask register is a single bit, so these instructions all operate on single vector registers regardless of the setting of the `vlmul` field in `vtype`. They do not change the value of `vlmul`. The destination vector register may be the same as either source vector register.

As with other vector instructions, the elements with indices less than `vstart` are unchanged, and `vstart` is reset to zero after execution. Vector mask logical instructions are always unmasked, so there are no inactive elements, and the encodings with `vm=0` are reserved. Mask elements past `vl`, the tail elements, are always updated with a tail-agnostic policy.

```
vmand.mm vd, vs2, vs1   # vd.mask[i] =   vs2.mask[i] &&  vs1.mask[i]
vmnand.mm vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] &&  vs1.mask[i])
vmandn.mm vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] && !vs1.mask[i]
vmxor.mm  vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] ^^  vs1.mask[i]
vmor.mm  vd, vs2, vs1   # vd.mask[i] =   vs2.mask[i] ||  vs1.mask[i]
vmnor.mm  vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] ||  vs1.mask[i])
vmorn.mm  vd, vs2, vs1  # vd.mask[i] =   vs2.mask[i] || !vs1.mask[i]
vmxnor.mm vd, vs2, vs1  # vd.mask[i] = !(vs2.mask[i] ^^  vs1.mask[i])
```

The previous assembler mnemonics `vmandnot` *and* `vmornot` *have been changed to* `vmandn` *and* `vmorn` *to be consistent with the equivalent scalar instructions. The old* `vmandnot` *and* `vmornot` *mnemonics can be retained as assembler aliases for compatibility.*

Several assembler pseudoinstructions are defined as shorthand for common uses of mask logical operations:

```
vmmv.m vd, vs  => vmand.mm vd, vs, vs   # Copy mask register
```

```
vmclr.m vd     => vmxor.mm vd, vd, vd   # Clear mask register
vmset.m vd     => vmxnor.mm vd, vd, vd  # Set mask register
vmnot.m vd, vs => vmnand.mm vd, vs, vs  # Invert bits
```

The `vmmv.m` instruction was previously called `vmcpy.m`, but with new layout it is more consistent to name as a "mv" because bits are copied without interpretation. The `vmcpy.m` assembler pseudoinstruction can be retained for compatibility. For implementations that internally rearrange bits according to EEW, a `vmmv.m` instruction with same source and destination can be used as idiom to force an internal reformat into a mask vector.

The set of eight mask logical instructions can generate any of the 16 possibly binary logical functions of the two input masks:

inputs				
0	0	1	1	src1
0	1	0	1	src2

output				instruction	pseudoinstruction
0	0	0	0	vmxor.mm vd, vd, vd	vmclr.m vd
1	0	0	0	vmnor.mm vd, src1, src2	
0	1	0	0	vmandn.mm vd, src2, src1	
1	1	0	0	vmnand.mm vd, src1, src1	vmnot.m vd, src1
0	0	1	0	vmandn.mm vd, src1, src2	
1	0	1	0	vmnand.mm vd, src2, src2	vmnot.m vd, src2
0	1	1	0	vmxor.mm vd, src1, src2	
1	1	1	0	vmnand.mm vd, src1, src2	
0	0	0	1	vmand.mm vd, src1, src2	
1	0	0	1	vmxnor.mm vd, src1, src2	
0	1	0	1	vmand.mm vd, src2, src2	vmmv.m vd, src2
1	1	0	1	vmorn.mm vd, src2, src1	
0	0	1	1	vmand.mm vd, src1, src1	vmmv.m vd, src1
1	0	1	1	vmorn.mm vd, src1, src2	
0	1	1	1	vmor.mm vd, src1, src2	
1	1	1	1	vmxnor.mm vd, vd, vd	vmset.m vd

The vector mask logical instructions are designed to be easily fused with a following masked vector operation to effectively expand the number of predicate registers by moving values into `v0` before use.

31.15.2. Vector count population in mask `vcpop.m`

```
vcpop.m rd, vs2, vm
```

This instruction previously had the assembler mnemonic `vpopc.m` but was renamed to be

> *consistent with the scalar instruction. The assembler instruction alias* `vpopc.m` *is being retained for software compatibility.*

The source operand is a single vector register holding mask register values as described in Section 31.4.5.

The `vcpop.m` instruction counts the number of mask elements of the active elements of the vector source mask register that have the value 1 and writes the result to a scalar `x` register.

The operation can be performed under a mask, in which case only the masked elements are counted.

```
vcpop.m rd, vs2, v0.t # x[rd] = sum_i ( vs2.mask[i] && v0.mask[i] )
```

The `vcpop.m` instruction writes `x[rd]` even if `vl`=0 (with the value 0, since no mask elements are active).

Traps on `vcpop.m` are always reported with a `vstart` of 0. The `vcpop.m` instruction will raise an illegal-instruction exception if `vstart` is non-zero.

31.15.3. `vfirst` find-first-set mask bit

```
vfirst.m rd, vs2, vm
```

The `vfirst` instruction finds the lowest-numbered active element of the source mask vector that has the value 1 and writes that element's index to a GPR. If no active element has the value 1, -1 is written to the GPR.

> *Software can assume that any negative value (highest bit set) corresponds to no element found, as vector lengths will never reach* $2^{(XLEN-1)}$ *on any implementation.*

The `vfirst.m` instruction writes `x[rd]` even if `vl`=0 (with the value -1, since no mask elements are active).

Traps on `vfirst` are always reported with a `vstart` of 0. The `vfirst` instruction will raise an illegal-instruction exception if `vstart` is non-zero.

31.15.4. `vmsbf.m` set-before-first mask bit

```
   vmsbf.m vd, vs2, vm

# Example

    7 6 5 4 3 2 1 0   Element number

    1 0 0 1 0 1 0 0   v3 contents
                      vmsbf.m v2, v3
    0 0 0 0 0 0 1 1   v2 contents

    1 0 0 1 0 1 0 1   v3 contents
                      vmsbf.m v2, v3
    0 0 0 0 0 0 0 0   v2
```

```
    0 0 0 0 0 0 0 0   v3 contents
                      vmsbf.m v2, v3
    1 1 1 1 1 1 1 1   v2

    1 1 0 0 0 0 1 1   v0 vcontents
    1 0 0 1 0 1 0 0   v3 contents
                      vmsbf.m v2, v3, v0.t
    0 1 x x x x 1 1   v2 contents
```

The `vmsbf.m` instruction takes a mask register as input and writes results to a mask register. The instruction writes a 1 to all active mask elements before the first active source element that is a 1, then writes a 0 to that element and all following active elements. If there is no set bit in the active elements of the source vector, then all active elements in the destination are written with a 1.

The tail elements in the destination mask register are updated under a tail-agnostic policy.

Traps on `vmsbf.m` are always reported with a `vstart` of 0. The `vmsbf` instruction will raise an illegal-instruction exception if `vstart` is non-zero.

The destination register cannot overlap the source register and, if masked, cannot overlap the mask register ('v0').

31.15.5. `vmsif.m` set-including-first mask bit

The vector mask set-including-first instruction is similar to set-before-first, except it also includes the element with a set bit.

```
   vmsif.m vd, vs2, vm

# Example

    7 6 5 4 3 2 1 0   Element number

    1 0 0 1 0 1 0 0   v3 contents
                      vmsif.m v2, v3
    0 0 0 0 0 1 1 1   v2 contents

    1 0 0 1 0 1 0 1   v3 contents
                      vmsif.m v2, v3
    0 0 0 0 0 0 0 1   v2

    1 1 0 0 0 0 1 1   v0 vcontents
    1 0 0 1 0 1 0 0   v3 contents
                      vmsif.m v2, v3, v0.t
    1 1 x x x x 1 1   v2 contents
```

The tail elements in the destination mask register are updated under a tail-agnostic policy.

Traps on `vmsif.m` are always reported with a `vstart` of 0. The `vmsif` instruction will raise an illegal-

instruction exception if `vstart` is non-zero.

The destination register cannot overlap the source register and, if masked, cannot overlap the mask register ('v0').

31.15.6. `vmsof.m` set-only-first mask bit

The vector mask set-only-first instruction is similar to set-before-first, except it only sets the first element with a bit set, if any.

```
    vmsof.m vd, vs2, vm

 # Example

     7 6 5 4 3 2 1 0   Element number

     1 0 0 1 0 1 0 0   v3 contents
                       vmsof.m v2, v3
     0 0 0 0 0 1 0 0   v2 contents

     1 0 0 1 0 1 0 1   v3 contents
                       vmsof.m v2, v3
     0 0 0 0 0 0 0 1   v2

     1 1 0 0 0 0 1 1   v0 vcontents
     1 1 0 1 0 1 0 0   v3 contents
                       vmsof.m v2, v3, v0.t
     0 1 x x x x 0 0   v2 contents
```

The tail elements in the destination mask register are updated under a tail-agnostic policy.

Traps on `vmsof.m` are always reported with a `vstart` of 0. The `vmsof` instruction will raise an illegal-instruction exception if `vstart` is non-zero.

The destination register cannot overlap the source register and, if masked, cannot overlap the mask register ('v0').

31.15.7. Example using vector mask instructions

The following is an example of vectorizing a data-dependent exit loop.

```
  # char* strcpy(char *dst, const char* src)
strcpy:
      mv a2, a0             # Copy dst
      li t0, -1             # Infinite AVL
loop:
    vsetvli x0, t0, e8, m8, ta, ma  # Max length vectors of bytes
    vle8ff.v v8, (a1)        # Get src bytes
      csrr t1, vl           # Get number of bytes fetched
```

```
    vmseq.vi v1, v8, 0      # Flag zero bytes
    vfirst.m a3, v1         # Zero found?
      add a1, a1, t1        # Bump pointer
    vmsif.m v0, v1          # Set mask up to and including zero byte.
    vse8.v v8, (a2), v0.t    # Write out bytes
      add a2, a2, t1        # Bump pointer
      bltz a3, loop         # Zero byte not found, so loop

      ret
```

```
  # char* strncpy(char *dst, const char* src, size_t n)
strncpy:
      mv a3, a0             # Copy dst
loop:
    vsetvli x0, a2, e8, m8, ta, ma   # Vectors of bytes.
    vle8ff.v v8, (a1)        # Get src bytes
    vmseq.vi v1, v8, 0      # Flag zero bytes
      csrr t1, vl           # Get number of bytes fetched
    vfirst.m a4, v1         # Zero found?
    vmsbf.m v0, v1          # Set mask up to before zero byte.
    vse8.v v8, (a3), v0.t    # Write out non-zero bytes
      bgez a4, zero_tail    # Zero remaining bytes.
      sub a2, a2, t1        # Decrement count.
      add a3, a3, t1        # Bump dest pointer
      add a1, a1, t1        # Bump src pointer
      bnez a2, loop         # Anymore?

      ret

zero_tail:
    sub a2, a2, a4          # Subtract count on non-zero bytes.
    add a3, a3, a4          # Advance past non-zero bytes.
    vsetvli t1, a2, e8, m8, ta, ma   # Vectors of bytes.
    vmv.v.i v0, 0           # Splat zero.

zero_loop:
    vse8.v v0, (a3)          # Store zero.
      sub a2, a2, t1        # Decrement count.
      add a3, a3, t1        # Bump pointer
      vsetvli t1, a2, e8, m8, ta, ma   # Vectors of bytes.
      bnez a2, zero_loop    # Anymore?

      ret
```

31.15.8. Vector Iota Instruction

The `viota.m` instruction reads a source vector mask register and writes to each element of the destination

vector register group the sum of all the bits of elements in the mask register whose index is less than the element, e.g., a parallel prefix sum of the mask values.

This instruction can be masked, in which case only the enabled elements contribute to the sum.

```
viota.m vd, vs2, vm

# Example

    7 6 5 4 3 2 1 0   Element number

    1 0 0 1 0 0 0 1   v2 contents
                      viota.m v4, v2 # Unmasked
    2 2 2 1 1 1 1 0   v4 result

    1 1 1 0 1 0 1 1   v0 contents
    1 0 0 1 0 0 0 1   v2 contents
    2 3 4 5 6 7 8 9   v4 contents
                      viota.m v4, v2, v0.t # Masked, vtype.vma=0
    1 1 1 5 1 7 1 0   v4 results
```

The result value is zero-extended to fill the destination element if SEW is wider than the result. If the result value would overflow the destination SEW, the least-significant SEW bits are retained.

Traps on `viota.m` are always reported with a `vstart` of 0, and execution is always restarted from the beginning when resuming after a trap handler. An illegal-instruction exception is raised if `vstart` is non-zero.

The destination register group cannot overlap the source register and, if masked, cannot overlap the mask register (`v0`).

The `viota.m` instruction can be combined with memory scatter instructions (indexed stores) to perform vector compress functions.

```
# Compact non-zero elements from input memory array to output memory array
#
# size_t compact_non_zero(size_t n, const int* in, int* out)
# {
#   size_t i;
#   int *p = out;
#
#   for (i=0; i<n; i++)
#   {
#       const int v = *in++;
#       if (v != 0)
#           *p++ = v;
#   }
#
#   return (size_t) (p - out);
# }
```

```
#
# a0 = n
# a1 = &in
# a2 = &out

compact_non_zero:
    li a6, 0                      # Clear count of non-zero elements
loop:
    vsetvli a5, a0, e32, m8, ta, ma   # 32-bit integers
    vle32.v v8, (a1)               # Load input vector
      sub a0, a0, a5               # Decrement number done
      slli a5, a5, 2               # Multiply by four bytes
    vmsne.vi v0, v8, 0             # Locate non-zero values
      add a1, a1, a5               # Bump input pointer
    vcpop.m a5, v0                 # Count number of elements set in v0
    viota.m v16, v0                # Get destination offsets of active elements
      add a6, a6, a5               # Accumulate number of elements
    vsll.vi v16, v16, 2, v0.t      # Multiply offsets by four bytes
      slli a5, a5, 2               # Multiply number of non-zero elements by
four bytes
    vsuxei32.v v8, (a2), v16, v0.t # Scatter using scaled viota results under
mask
      add a2, a2, a5               # Bump output pointer
      bnez a0, loop                # Any more?

      mv a0, a6                    # Return count
      ret
```

31.15.9. Vector Element Index Instruction

The `vid.v` instruction writes each element's index to the destination vector register group, from 0 to `vl`-1.

```
vid.v vd, vm  # Write element ID to destination.
```

The instruction can be masked. Masking does not change the index value written to active elements.

The `vs2` field of the instruction must be set to `v0`, otherwise the encoding is *reserved*.

The result value is zero-extended to fill the destination element if SEW is wider than the result. If the result value would overflow the destination SEW, the least-significant SEW bits are retained.

Microarchitectures can implement `vid.v` *instruction using the same datapath as* `viota.m` *but with an implicit set mask source.*

31.16. Vector Permutation Instructions

A range of permutation instructions are provided to move elements around within the vector registers.

31.16.1. Integer Scalar Move Instructions

The integer scalar read/write instructions transfer a single value between a scalar `x` register and element 0 of a vector register. The instructions ignore LMUL and vector register groups.

```
vmv.x.s rd, vs2  # x[rd] = vs2[0] (vs1=0)
vmv.s.x vd, rs1  # vd[0] = x[rs1] (vs2=0)
```

The `vmv.x.s` instruction copies a single SEW-wide element from index 0 of the source vector register to a destination integer register. If SEW > XLEN, the least-significant XLEN bits are transferred and the upper SEW-XLEN bits are ignored. If SEW < XLEN, the value is sign-extended to XLEN bits.

`vmv.x.s` *performs its operation even if* `vstart` ≥ `vl` *or* `vl`=0.

The `vmv.s.x` instruction copies the scalar integer register to element 0 of the destination vector register. If SEW < XLEN, the least-significant bits are copied and the upper XLEN-SEW bits are ignored. If SEW > XLEN, the value is sign-extended to SEW bits. The other elements in the destination vector register (0 < index < VLEN/SEW) are treated as tail elements using the current tail agnostic/undisturbed policy. If `vstart` ≥ `vl`, no operation is performed and the destination register is not updated.

As a consequence, when `vl`=0, *no elements are updated in the destination vector register group, regardless of* `vstart`.

The encodings corresponding to the masked versions (`vm=0`) of `vmv.x.s` and `vmv.s.x` are reserved.

31.16.2. Floating-Point Scalar Move Instructions

The floating-point scalar read/write instructions transfer a single value between a scalar `f` register and element 0 of a vector register. The instructions ignore LMUL and vector register groups.

```
vfmv.f.s rd, vs2  # f[rd] = vs2[0] (rs1=0)
vfmv.s.f vd, rs1  # vd[0] = f[rs1] (vs2=0)
```

The `vfmv.f.s` instruction copies a single SEW-wide element from index 0 of the source vector register to a destination scalar floating-point register.

`vfmv.f.s` *performs its operation even if* `vstart` ≥ `vl` *or* `vl`=0.

The `vfmv.s.f` instruction copies the scalar floating-point register to element 0 of the destination vector register. The other elements in the destination vector register (0 < index < VLEN/SEW) are treated as tail elements using the current tail agnostic/undisturbed policy. If `vstart` ≥ `vl`, no operation is performed and the destination register is not updated.

As a consequence, when `vl`=0, *no elements are updated in the destination vector register group, regardless of* `vstart`.

The encodings corresponding to the masked versions (`vm=0`) of `vfmv.f.s` and `vfmv.s.f` are reserved.

31.16.3. Vector Slide Instructions

The slide instructions move elements up and down a vector register group.

The slide operations can be implemented much more efficiently than using the arbitrary register gather instruction. Implementations may optimize certain OFFSET values for `vslideup` *and* `vslidedown`*. In particular, power-of-2 offsets may operate substantially faster than other offsets.*

For all of the `vslideup`, `vslidedown`, `v[f]slide1up`, and `v[f]slide1down` instructions, if `vstart` ≥ `vl`, the instruction performs no operation and leaves the destination vector register unchanged.

As a consequence, when `vl`*=0, no elements are updated in the destination vector register group, regardless of* `vstart`*.*

The tail agnostic/undisturbed policy is followed for tail elements.

The slide instructions may be masked, with mask element *i* controlling whether *destination* element *i* is written. The mask undisturbed/agnostic policy is followed for inactive elements.

31.16.3.1. Vector Slide-up Instructions

```
vslideup.vx vd, vs2, rs1, vm        # vd[i+x[rs1]] = vs2[i]
vslideup.vi vd, vs2, uimm, vm       # vd[i+uimm] = vs2[i]
```

For `vslideup`, the value in `vl` specifies the maximum number of destination elements that are written. The start index (*OFFSET*) for the destination can be either specified using an unsigned integer in the `x` register specified by `rs1`, or a 5-bit immediate, zero-extended to XLEN bits. If XLEN > SEW, *OFFSET* is *not* truncated to SEW bits. Destination elements *OFFSET* through `vl`-1 are written if unmasked and if *OFFSET* < `vl`.

```
vslideup behavior for destination elements (vstart < vl)

OFFSET is amount to slideup, either from x register or a 5-bit immediate

                 0 <= i < min(vl, max(vstart, OFFSET))  Unchanged
max(vstart, OFFSET) <= i < vl                           vd[i] = vs2[i-OFFSET]
if v0.mask[i] enabled
                vl <= i < VLMAX                         Follow tail policy
```

The destination vector register group for `vslideup` cannot overlap the source vector register group, otherwise the instruction encoding is reserved.

The non-overlap constraint avoids WAR hazards on the input vectors during execution, and enables restart with non-zero `vstart`*.*

31.16.3.2. Vector Slide-down Instructions

```
vslidedown.vx vd, vs2, rs1, vm       # vd[i] = vs2[i+x[rs1]]
vslidedown.vi vd, vs2, uimm, vm      # vd[i] = vs2[i+uimm]
```

For `vslidedown`, the value in `vl` specifies the maximum number of destination elements that are written.

The remaining elements past `vl` are handled according to the current tail policy (Section 31.3.4.3).

The start index (*OFFSET*) for the source can be either specified using an unsigned integer in the `x` register specified by `rs1`, or a 5-bit immediate, zero-extended to XLEN bits. If XLEN > SEW, *OFFSET* is *not* truncated to SEW bits.

```
vslidedown behavior for source elements for element i in slide (vstart < vl)
               0 <= i+OFFSET < VLMAX   src[i] = vs2[i+OFFSET]
           VLMAX <= i+OFFSET           src[i] = 0

vslidedown behavior for destination element i in slide (vstart < vl)
                0 <= i < vstart         Unchanged
           vstart <= i < vl             vd[i] = src[i] if v0.mask[i] enabled
               vl <= i < VLMAX          Follow tail policy
```

31.16.3.3. Vector Slide-1-up

Variants of slide are provided that only move by one element but which also allow a scalar integer value to be inserted at the vacated element position.

```
vslide1up.vx  vd, vs2, rs1, vm        # vd[0]=x[rs1], vd[i+1] = vs2[i]
```

The `vslide1up` instruction places the `x` register argument at location 0 of the destination vector register group, provided that element 0 is active, otherwise the destination element update follows the current mask agnostic/undisturbed policy. If XLEN < SEW, the value is sign-extended to SEW bits. If XLEN > SEW, the least-significant bits are copied over and the high XLEN-SEW bits are ignored.

The remaining active `vl`-1 elements are copied over from index *i* in the source vector register group to index *i*+1 in the destination vector register group.

The `vl` register specifies the maximum number of destination vector register elements updated with source values, and remaining elements past `vl` are handled according to the current tail policy (Section 31.3.4.3).

```
vslide1up behavior when vl > 0

                 i < vstart  unchanged
             0 = i = vstart  vd[i] = x[rs1] if v0.mask[i] enabled
max(vstart, 1) <= i < vl     vd[i] = vs2[i-1] if v0.mask[i] enabled
            vl <= i < VLMAX  Follow tail policy
```

The `vslide1up` instruction requires that the destination vector register group does not overlap the source vector register group. Otherwise, the instruction encoding is reserved.

31.16.3.4. Vector Floating-Point Slide-1-up Instruction

```
vfslide1up.vf vd, vs2, rs1, vm        # vd[0]=f[rs1], vd[i+1] = vs2[i]
```

The `vfslide1up` instruction is defined analogously to `vslide1up`, but sources its scalar argument from an `f` register.

31.16.3.5. Vector Slide-1-down Instruction

The `vslide1down` instruction copies the first `vl`-1 active elements values from index *i*+1 in the source vector register group to index *i* in the destination vector register group.

The `vl` register specifies the maximum number of destination vector register elements written with source values, and remaining elements past `vl` are handled according to the current tail policy (Section 31.3.4.3).

```
vslide1down.vx  vd, vs2, rs1, vm      # vd[i] = vs2[i+1], vd[vl-1]=x[rs1]
```

The `vslide1down` instruction places the `x` register argument at location `vl`-1 in the destination vector register, provided that element `vl-1` is active, otherwise the destination element update follows the current mask agnostic/undisturbed policy. If XLEN < SEW, the value is sign-extended to SEW bits. If XLEN > SEW, the least-significant bits are copied over and the high SEW-XLEN bits are ignored.

```
vslide1down behavior

          i < vstart  unchanged
vstart <= i < vl-1    vd[i] = vs2[i+1] if v0.mask[i] enabled
vstart <= i = vl-1    vd[vl-1] = x[rs1] if v0.mask[i] enabled
    vl <= i < VLMAX   Follow tail policy
```

The `vslide1down` instruction can be used to load values into a vector register without using memory and without disturbing other vector registers. This provides a path for debuggers to modify the contents of a vector register, albeit slowly, with multiple repeated `vslide1down` invocations.

31.16.3.6. Vector Floating-Point Slide-1-down Instruction

```
vfslide1down.vf vd, vs2, rs1, vm      # vd[i] = vs2[i+1], vd[vl-1]=f[rs1]
```

The `vfslide1down` instruction is defined analogously to `vslide1down`, but sources its scalar argument from an `f` register.

31.16.4. Vector Register Gather Instructions

The vector register gather instructions read elements from a first source vector register group at locations given by a second source vector register group. The index values in the second vector are treated as unsigned integers. The source vector can be read at any index < VLMAX regardless of `vl`. The maximum number of elements to write to the destination register is given by `vl`, and the remaining elements past `vl` are handled according to the current tail policy (Section 31.3.4.3). The operation can be masked, and the mask undisturbed/agnostic policy is followed for inactive elements.

```
vrgather.vv vd, vs2, vs1, vm     # vd[i] = (vs1[i] >= VLMAX) ? 0 : vs2[vs1[i]];
```

```
vrgatherei16.vv vd, vs2, vs1, vm # vd[i] = (vs1[i] >= VLMAX) ? 0 : vs2[vs1[i]];
```

The `vrgather.vv` form uses SEW/LMUL for both the data and indices. The `vrgatherei16.vv` form uses SEW/LMUL for the data in `vs2` but EEW=16 and EMUL = (16/SEW)*LMUL for the indices in `vs1`.

When SEW=8, `vrgather.vv` can only reference vector elements 0-255. The `vrgatherei16` form can index 64K elements, and can also be used to reduce the register capacity needed to hold indices when SEW > 16.

If an element index is out of range (`vs1[i]` ≥ VLMAX) then zero is returned for the element value.

Vector-scalar and vector-immediate forms of the register gather are also provided. These read one element from the source vector at the given index, and write this value to the active elements of the destination vector register. The index value in the scalar register and the immediate, zero-extended to XLEN bits, are treated as unsigned integers. If XLEN > SEW, the index value is *not* truncated to SEW bits.

These forms allow any vector element to be "splatted" to an entire vector.

```
vrgather.vx vd, vs2, rs1, vm  # vd[i] = (x[rs1] >= VLMAX) ? 0 : vs2[x[rs1]]
vrgather.vi vd, vs2, uimm, vm # vd[i] =  (uimm >= VLMAX)  ? 0 : vs2[uimm]
```

For any `vrgather` instruction, the destination vector register group cannot overlap with the source vector register groups, otherwise the instruction encoding is reserved.

31.16.5. Vector Compress Instruction

The vector compress instruction allows elements selected by a vector mask register from a source vector register group to be packed into contiguous elements at the start of the destination vector register group.

```
vcompress.vm vd, vs2, vs1  # Compress into vd elements of vs2 where vs1 is
enabled
```

The vector mask register specified by `vs1` indicates which of the first `vl` elements of vector register group `vs2` should be extracted and packed into contiguous elements at the beginning of vector register `vd`. The remaining elements of `vd` are treated as tail elements according to the current tail policy (Section 31.3.4.3).

```
Example use of vcompress instruction

8 7 6 5 4 3 2 1 0   Element number

1 1 0 1 0 0 1 0 1   v0
8 7 6 5 4 3 2 1 0   v1
1 2 3 4 5 6 7 8 9   v2
                        vsetivli     t0, 9, e8, m1, tu, ma
                        vcompress.vm v2, v1, v0
1 2 3 4 8 7 5 2 0   v2
```

T`vcompress` is encoded as an unmasked instruction (`vm=1`). The equivalent masked instruction (`vm=0`) is reserved.

The destination vector register group cannot overlap the source vector register group or the source mask register, otherwise the instruction encoding is reserved.

A trap on a `vcompress` instruction is always reported with a `vstart` of 0. Executing a `vcompress` instruction with a non-zero `vstart` raises an illegal-instruction exception.

Although possible, `vcompress` *is one of the more difficult instructions to restart with a non-zero* `vstart`*, so assumption is implementations will choose not do that but will instead restart from element 0. This does mean elements in destination register after* `vstart` *will already have been updated.*

31.16.5.1. Synthesizing `vdecompress`

There is no inverse `vdecompress` provided, as this operation can be readily synthesized using iota and a masked vrgather:

```
Desired functionality of 'vdecompress'
7 6 5 4 3 2 1 0     # vid

      e d c b a     # packed vector of 5 elements
1 0 0 1 1 1 0 1     # mask vector of 8 elements
p q r s t u v w     # destination register before vdecompress

e q r d c b v a     # result of vdecompress
```

```
# v0 holds mask
# v1 holds packed data
# v11 holds input expanded vector and result
viota.m v10, v0                 # Calc iota from mask in v0
vrgather.vv v11, v1, v10, v0.t  # Expand into destination
```

```
p q r s t u v w    # v11 destination register
      e d c b a    # v1 source vector
1 0 0 1 1 1 0 1    # v0 mask vector

4 4 4 3 2 1 1 0    # v10 result of viota.m
e q r d c b v a    # v11 destination after vrgather using viota.m under mask
```

31.16.6. Whole Vector Register Move

The `vmv<nr>r.v` instructions copy whole vector registers (i.e., all VLEN bits) and can copy whole vector register groups. The `nr` value in the opcode is the number of individual vector registers, NREG, to copy. The instructions operate as if EEW=SEW, EMUL = NREG, effective length `evl`= EMUL * VLEN/SEW.

These instructions are intended to aid compilers to shuffle vector registers without needing to know or change `vl`.

The usual property that no elements are written if `vstart` ≥ `vl` does not apply to these instructions. Similarly, the property that the instructions are reserved if `vstart` exceeds the largest element index for the current `vtype` setting does not apply. Instead, the instructions are reserved if `vstart` ≥ `evl`.

If `vd` *is equal to* `vs2`*, the instruction does not change any vector register state. Implementations that rearrange data internally can treat this instruction as a hint that the register group will next be accessed with an EEW equal to SEW.*

The instruction is encoded as an OPIVI instruction. The number of vector registers to copy is encoded in the low three bits of the `simm` field (`simm[2:0]`) using the same encoding as the `nf[2:0]` field for memory instructions (Figure Table 55), i.e., `simm[2:0]` = NREG-1.

The value of NREG must be 1, 2, 4, or 8, and values of `simm[4:0]` other than 0, 1, 3, and 7 are reserved.

A future extension may support other numbers of registers to be moved.

The instruction uses the same funct6 encoding as the `vsmul` *instruction but with an immediate operand, and only the unmasked version (*`vm=1`*). This encoding is chosen as it is close to the related* `vmerge` *encoding, and it is unlikely the* `vsmul` *instruction would benefit from an immediate form.*

```
vmv<nr>r.v vd, vs2  # General form

vmv1r.v v1, v2   #  Copy v1=v2
vmv2r.v v10, v12 #  Copy v10=v12; v11=v13
vmv4r.v v4, v8   #  Copy v4=v8; v5=v9; v6=v10; v7=v11
vmv8r.v v0, v8   #  Copy v0=v8; v1=v9; ...;  v7=v15
```

The source and destination vector register numbers must be aligned appropriately for the vector register group size, and encodings with other vector register numbers are reserved.

A future extension may relax the vector register alignment restrictions.

31.17. Exception Handling

On a trap during a vector instruction (caused by either a synchronous exception or an asynchronous interrupt), the existing `*epc` CSR is written with a pointer to the trapping vector instruction, while the `vstart` CSR contains the element index on which the trap was taken.

We chose to add a `vstart` *CSR to allow resumption of a partially executed vector instruction to reduce interrupt latencies and to simplify forward-progress guarantees. This is similar to the scheme in the IBM 3090 vector facility. To ensure forward progress without the* `vstart` *CSR, implementations would have to guarantee an entire vector instruction can always complete atomically without generating a trap. This is particularly difficult to ensure in the presence of constant-stride or scatter/gather operations and demand-paged virtual memory.*

31.17.1. Precise vector traps

We assume most supervisor-mode environments with demand-paging will require precise

vector traps.

Precise vector traps require that:

1. all instructions older than the trapping vector instruction have committed their results
2. no instructions newer than the trapping vector instruction have altered architectural state
3. any operations within the trapping vector instruction affecting result elements preceding the index in the `vstart` CSR have committed their results
4. no operations within the trapping vector instruction affecting elements at or following the `vstart` CSR have altered architectural state except if restarting and completing the affected vector instruction will nevertheless produce the correct final state.

We relax the last requirement to allow elements following `vstart` to have been updated at the time the trap is reported, provided that re-executing the instruction from the given `vstart` will correctly overwrite those elements.

In idempotent memory regions, vector store instructions may have updated elements in memory past the element causing a synchronous trap. Non-idempotent memory regions must not have been updated for indices equal to or greater than the element that caused a synchronous trap during a vector store instruction.

Except where noted above, vector instructions are allowed to overwrite their inputs, and so in most cases, the vector instruction restart must be from the `vstart` element index. However, there are a number of cases where this overwrite is prohibited to enable execution of the vector instructions to be idempotent and hence restartable from an earlier index location.

Implementations must ensure forward progress can be eventually guaranteed for the element or segment reported by `vstart`.

31.17.2. Imprecise vector traps

Imprecise vector traps are traps that are not precise. In particular, instructions newer than `*epc` may have committed results, and instructions older than `*epc` may have not completed execution. Imprecise traps are primarily intended to be used in situations where reporting an error and terminating execution is the appropriate response.

A profile might specify that interrupts are precise while other traps are imprecise. We assume many embedded implementations will generate only imprecise traps for vector instructions on fatal errors, as they will not require resumable traps.

Imprecise traps shall report the faulting element in `vstart` for traps caused by synchronous vector exceptions.

There is no support for imprecise traps in the current standard extensions.

31.17.3. Selectable precise/imprecise traps

Some profiles may choose to provide a privileged mode bit to select between precise and imprecise vector traps. Imprecise mode would run at high-performance but possibly make it difficult to discern error causes, while precise mode would run more slowly, but support debugging of errors albeit with a possibility of not experiencing the same errors as in imprecise mode.

This mechanism is not defined in the current standard extensions.

31.17.4. Swappable traps

Another trap mode can support swappable state in the vector unit, where on a trap, special instructions can save and restore the vector unit microarchitectural state, to allow execution to continue correctly around imprecise traps.

This mechanism is not defined in the current standard extensions.

A future extension might define a standard way of saving and restoring opaque microarchitectural state from a vector unit implementation to support context switching with imprecise traps.

31.18. Standard Vector Extensions

This section describes the standard vector extensions. A set of smaller extensions intended for embedded use are named with a "Zve" prefix, while a larger vector extension designed for application processors is named as a single-letter V extension. A set of vector length extension names with prefix "Zvl" are also provided.

The initial vector extensions are designed to act as a base for additional vector extensions in various domains, including cryptography and machine learning.

31.18.1. Zvl*: Minimum Vector Length Standard Extensions

All standard vector extensions have a minimum required VLEN as described below. A set of vector length extensions are provided to increase the minimum vector length of a vector extension.

The vector length extensions can be used to either specify additional software or architecture profile requirements, or to advertise hardware capabilities.

Table 59. Vector length extensions

Extension	Minimum VLEN
Zvl32b	32
Zvl64b	64
Zvl128b	128
Zvl256b	256
Zvl512b	512
Zvl1024b	1024

Longer vector length extensions should follow the same pattern.

Every vector length extension effectively includes all shorter vector length extensions.

Explicit use of the Zvl32b extension string is not required for any standard vector extension as they all effectively mandate at least this minimum, but the string can be useful when stating hardware capabilities.

31.18.2. Zve*: Vector Extensions for Embedded Processors

The following five standard extensions are defined to provide varying degrees of vector support and are intended for use with embedded processors. Any of these extensions can be added to base ISAs with

XLEN=32 or XLEN=64. The table lists the minimum VLEN and supported EEWs for each extension as well as what floating-point types are supported.

Table 60. Embedded vector extensions

Extension	Minimum VLEN	Supported EEW	FP32	FP64
Zve32x	32	8, 16, 32	N	N
Zve32f	32	8, 16, 32	Y	N
Zve64x	64	8, 16, 32, 64	N	N
Zve64f	64	8, 16, 32, 64	Y	N
Zve64d	64	8, 16, 32, 64	Y	Y

The Zve32f and Zve64x extensions depend on the Zve32x extension. The Zve64f extension depends on the Zve32f and Zve64x extensions. The Zve64d extension depends on the Zve64f extension.

All Zve* extensions have precise traps.

There is currently no standard support for handling imprecise traps, so standard extensions have to provide precise traps.

All Zve* extensions provide support for EEW of 8, 16, and 32, and Zve64* extensions also support EEW of 64.

All Zve* extensions support the vector configuration instructions (Section 31.6).

All Zve* extensions support all vector load and store instructions (Section 31.7), except Zve64* extensions do not support EEW=64 for index values when XLEN=32.

All Zve* extensions support all vector integer instructions (Section 31.11), except that the `vmulh` integer multiply variants that return the high word of the product (`vmulh.vv`, `vmulh.vx`, `vmulhu.vv`, `vmulhu.vx`, `vmulhsu.vv`, `vmulhsu.vx`) are not included for EEW=64 in Zve64*.

Producing the high-word of a product can take substantial additional gates for large EEW.

All Zve* extensions support all vector fixed-point arithmetic instructions (Section 31.12), except that `vsmul.vv` and `vsmul.vx` are not included in EEW=64 in Zve64*.

As with `vmulh`, `vsmul` requires a large amount of additional logic, and 64-bit fixed-point multiplies are relatively rare.

All Zve* extensions support all vector integer single-width and widening reduction operations (Section 31.14.1, Section 31.14.2).

All Zve* extensions support all vector mask instructions (Section 31.15).

All Zve* extensions support all vector permutation instructions (Section 31.16), except that Zve32x and Zve64x do not include those with floating-point operands, and Zve64f does not include those with EEW=64 floating-point operands.

The Zve32x extension depends on the Zicsr extension. The Zve32f and Zve64f extensions depend upon the F extension, and implement all vector floating-point instructions (Section 31.13) for floating-point operands with EEW=32. Vector single-width floating-point reduction operations (Section 31.14.3) for EEW=32 are supported.

The Zve64d extension depends upon the D extension, and implements all vector floating-point

instructions (Section 31.13) for floating-point operands with EEW=32 or EEW=64 (including widening instructions and conversions between FP32 and FP64). Vector single-width floating-point reductions (Section 31.14.3) for EEW=32 and EEW=64 are supported as well as widening reductions from FP32 to FP64.

31.18.3. V: Vector Extension for Application Processors

The single-letter V extension is intended for use in application processor profiles.

The `misa.v` bit is set for implementations providing `misa` and supporting V.

The V vector extension has precise traps.

The V vector extension depends upon the Zvl128b and Zve64d extensions.

The value of 128 was chosen as a compromise for application processors. Providing a larger VLEN allows strip-mining code to be elided in some cases for short vectors, but also increases the size of the minimum implementation. Note that larger LMUL can be used to avoid strip mining for longer known-size application vectors at the cost of having fewer available vector register groups. For example, an LMUL of 8 allows vectors of up to sixteen 64-bit elements to be processed without strip mining using four vector register groups.

The V extension supports EEW of 8, 16, and 32, and 64.

The V extension supports the vector configuration instructions (Section 31.6).

The V extension supports all vector load and store instructions (Section 31.7), except the V extension does not support EEW=64 for index values when XLEN=32.

The V extension supports all vector integer instructions (Section 31.11).

The V extension supports all vector fixed-point arithmetic instructions (Section 31.12).

The V extension supports all vector integer single-width and widening reduction operations (Section 31.14.1, Section 31.14.2).

The V extension supports all vector mask instructions (Section 31.15).

The V extension supports all vector permutation instructions (Section 31.16).

The V extension depends upon the F and D extensions, and implements all vector floating-point instructions (Section 31.13) for floating-point operands with EEW=32 or EEW=64 (including widening instructions and conversions between FP32 and FP64). Vector single-width floating-point reductions (Section 31.14.3) for EEW=32 and EEW=64 are supported as well as widening reductions from FP32 to FP64.

As is the case with other RISC-V extensions, it is valid to include overlapping extensions in the same ISA string. For example, RV64GCV and RV64GCV_Zve64f are both valid and equivalent ISA strings, as is RV64GCV_Zve64f_Zve32x_Zvl128b.

31.18.4. Zvfhmin: Vector Extension for Minimal Half-Precision Floating-Point

The Zvfhmin extension provides minimal support for vectors of IEEE 754-2008 binary16 values, adding conversions to and from binary32. When the Zvfhmin extension is implemented, the `vfwcvt.f.f.v` and `vfncvt.f.f.w` instructions become defined when SEW=16. The EEW=16 floating-point operands of these

instructions use the binary16 format.

The Zvfhmin extension depends on the Zve32f extension.

31.18.5. Zvfh: Vector Extension for Half-Precision Floating-Point

The Zvfh extension provides support for vectors of IEEE 754-2008 binary16 values. When the Zvfh extension is implemented, all instructions in Section 31.13, Section 31.14.3, Section 31.14.4, Section 31.16.2, Section 31.16.3.4, and Section 31.16.3.6 become defined when SEW=16. The EEW=16 floating-point operands of these instructions use the binary16 format.

Additionally, conversions between 8-bit integers and binary16 values are provided. The floating-point-to-integer narrowing conversions (`vfncvt[.rtz].x[u].f.w`) and integer-to-floating-point widening conversions (`vfwcvt.f.x[u].v`) become defined when SEW=8.

The Zvfh extension depends on the Zve32f and Zfhmin extensions.

Requiring basic scalar half-precision support makes Zvfh's vector-scalar instructions substantially more useful. We considered requiring more complete scalar half-precision support, but we reasoned that, for many half-precision vector workloads, performing the scalar computation in single-precision will suffice.

31.19. Vector Element Groups

Some vector instructions treat operands as a vector of one or more *element groups*, where each element group is a fixed number of elements. For example, complex numbers can be viewed as a two-element group (one real element and one imaginary element). As another example, the SHA-256 cryptographic instructions in the Zvknha extension operate on 128-bit values represented as a 4-element group of 32-bit elements.

This section describes recommendations and terminology for generic instruction set design for vector instructions that operate on element groups.

31.19.1. Element Group Size

The *element group size* (EGS) is the number of elements in one group, and must be a power-of-two (POT).

Support for non-POT EGS was considered but causes many practical complications and so has been dropped. Error checking for `vl` is a little more difficult. For LMUL>1, non-POT EGSs will result in groups straddling the individual vector registers in a vector register group. Non-POT EGS can also cause large increases in the lowest-common-multiple of element group sizes, which adds constraints to `vl` setting in order to avoid splitting an element group across strip-mine iterations in vector-length-agnostic code.

The element group size is statically encoded in the instruction, often implicitly as part of the opcode.

Vector instructions with EGS > VLMAX are reserved.

The vector instructions in the base V vector ISA can be viewed as all having an element group size of 1 for all operands statically encoded in the instruction.

Many operations only make sense with a certain number of elements per group (e.g., complex operations require a element group size of 2 and SHA-256 requires an element group size of 4).

31.19.2. Setting `vl`

Each source and destination operand to a vector instruction might be defined as either a single element group or a vector of element groups. When an operand is a vector of element groups, the `vl` setting must correspond to an integer multiple of the element group size, with other values of `vl` reserved.

For example, a SHA-256 instruction would require that `vl` is a multiple of 4.

When element group instructions are present, an additional constraint is placed on the setting of `vl` based on an AVL value (augmenting Section 31.6.3). EGSMAX is the largest EGS supported by the implementation. When AVL > VLMAX, the value of `vl` must be set to either VLMAX or a positive integer multiple of EGSMAX.

As the base vector extension only has element group size of 1, this constraint is backwards-compatible.

This constraint prevents element groups being broken across strip-mining iterations in vector-length-agnostic code when a VLMAX-size vector would otherwise be able to accommodate a whole number of element groups.

If EEW is encoded statically in the instruction, or if an instruction has multiple operands containing vectors of element groups with different EEW, an appropriate SEW must be chosen for `vsetvl` instructions.

Additional constraints may be required for some element group instructions to ensure legal length values for all operands.

31.19.3. Determining EEW

The `vtype` SEW can be used to indicate or calculate the effective element size (EEW) of one or more operands of an element group instruction. Where the operand is an element group, SEW and EEW refer to the number of bits in each individual element within a group not the number of bits in the group as a whole.

Alternatively, the opcode might encode EEW of all operands statically and ignore the value of SEW when the operation only makes sense for a single size on each operand.

Many operations are only defined for one EEW, e.g., SHA-256 requires EEW=32. Encoding EEWs statically in the instruction removes a dynamic dependency on the SEW value and the need to check for errors in SEW values. However, ignoring SEW also prevents reuse of the static opcode with a different dynamic SEW, and in many cases, the SEW setting will be needed for regular vector instructions used to process the individual elements in the vector.

31.19.4. Determining EMUL

The `vtype` LMUL setting can be used to indicate or calculate the effective length multiplier (EMUL) for one or more operands. Element group instructions tend to exhibit a much wider range of relationships between various operand EEW/EMUL values. For example, an instruction might take a vector of length N of 4-element groups with EEW=8b and reduce each group to produce a vector length N of 1-element groups with EEW=32b. In this case, the input and output EMUL values are equal even though the EEW settings differ by a factor of 4.

Each source and destination operand to a vector instruction may have a different element group size, different EMUL, and/or different EEW.

31.19.5. Element Group Width

The *element group width* (EGW) is the number of bits in the element group as a whole. For example, the SHA-256 instructions in the Zvknha extension operate on an EGW of 128, with EGS=4 and EEW=32. It is possible to use LMUL to concatenate multiple vector registers together to support larger EGW>VLEN.

If software using large-EGW instructions need be portable across a range of implementations, some of which may have VLEN<EGW and hence require LMUL>1, then software can only use a subset of the architectural registers. Profiles can set minimum VLEN requirements to inform authors of such software.

Element group operations by their nature will gather data from across a wider portion of a vector datapath than regular vector instructions. Some element group instructions might allow temporal execution of individual element operations in a larger group, while others will require all EGW bits of a group to be presented to a functional unit at the same time.

31.19.6. Masking

No ratified extensions include masked element-group instructions. Future extensions might extend the element-group scheme to support element-level masking, or might define the concept of a *mask element group* (which might, e.g., update the destination element group if any mask bit in the mask element group is set).

31.20. Vector Instruction Listing

Integer					Integer				FP			
funct3					funct3				funct3			
OPIVV	V				OPMVV	V			OPFVV	V		
OPIVX		X			OPMVX		X		OPFVF		F	
OPIVI			I									

funct6					funct6				funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd
000001					000001	V		vredand	000001	V		vfredusum
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub
000011		X	I	vrsub	000011	V		vredxor	000011	V		vfredosum
000100	V	X		vminu	000100	V		vredminu	000100	V	F	vfmin
000101	V	X		vmin	000101	V		vredmin	000101	V		vfredmin
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F	vfmax

funct6					funct6				funct6			
000111	V	X		vmax	000111	V		vredmax	000111	V		vfredmax
001000					001000	V	X	vaaddu	001000	V	F	vfsgnj
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F	vfsgnjn
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F	vfsgnjx
001011	V	X	I	vxor	001011	V	X	vasub	001011			
001100	V	X	I	vrgather	001100				001100			
001101					001101				001101			
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up
001110	V			vrgatherei16								
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

funct6					funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXUNARY0	010000	V		VWFUNARY0
					010000		X	VRXUNARY0	010000		F	VRFUNARY0
010001	V	X	I	vmadc	010001				010001			
010010	V	X		vsbc	010010	V		VXUNARY0	010010	V		VFUNARY0
010011	V	X		vmsbc	010011				010011	V		VFUNARY1
010100					010100	V		VMUNARY0	010100			
010101					010101				010101			
010110					010110				010110			
010111	V	X	I	vmerge/vmv	010111	V		vcompress	010111		F	vfmerge/vfmv
011000	V	X	I	vmseq	011000	V		vmandn	011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		vmand	011001	V	F	vmfle
011010	V	X		vmsltu	011010	V		vmor	011010			
011011	V	X		vmslt	011011	V		vmxor	011011	V	F	vmflt
011100	V	X	I	vmsleu	011100	V		vmorn	011100	V	F	vmfne
011101	V	X	I	vmsle	011101	V		vmnand	011101		F	vmfgt
011110		X	I	vmsgtu	011110	V		vmnor	011110			

funct6					funct6				funct6			
011111		X	I	vmsgt	011111	V		vmxnor	011111		F	vmfge

funct6					funct6				funct6			
100000	V	X	I	vsaddu	100000	V	X	vdivu	100000	V	F	vfdiv
100001	V	X	I	vsadd	100001	V	X	vdiv	100001		F	vfrdiv
100010	V	X		vssubu	100010	V	X	vremu	100010			
100011	V	X		vssub	100011	V	X	vrem	100011			
100100					100100	V	X	vmulhu	100100	V	F	vfmul
100101	V	X	I	vsll	100101	V	X	vmul	100101			
100110					100110	V	X	vmulhsu	100110			
100111	V	X		vsmul	100111	V	X	vmulh	100111		F	vfrsub
100111			I	vmv<nr>r								
101000	V	X	I	vsrl	101000				101000	V	F	vfmadd
101001	V	X	I	vsra	101001	V	X	vmadd	101001	V	F	vfnmadd
101010	V	X	I	vssrl	101010				101010	V	F	vfmsub
101011	V	X	I	vssra	101011	V	X	vnmsub	101011	V	F	vfnmsub
101100	V	X	I	vnsrl	101100				101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	vmacc	101101	V	F	vfnmacc
101110	V	X	I	vnclipu	101110				101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	vnmsac	101111	V	F	vfnmsac

funct6					funct6				funct6			
110000	V			vwredsumu	110000	V	X	vwaddu	110000	V	F	vfwadd
110001	V			vwredsum	110001	V	X	vwadd	110001	V		vfwredusum
110010					110010	V	X	vwsubu	110010	V	F	vfwsub
110011					110011	V	X	vwsub	110011	V		vfwredosum
110100					110100	V	X	vwaddu.w	110100	V	F	vfwadd.w
110101					110101	V	X	vwadd.w	110101			

funct6					funct6				funct6			
110110					110110	V	X	vwsubu.w	110110	V	F	vfwsub.w
110111					110111	V	X	vwsub.w	110111			
111000					111000	V	X	vwmulu	111000	V	F	vfwmul
111001					111001				111001			
111010					111010	V	X	vwmulsu	111010			
111011					111011	V	X	vwmul	111011			
111100					111100	V	X	vwmaccu	111100	V	F	vfwmacc
111101					111101	V	X	vwmacc	111101	V	F	vfwnmacc
111110					111110		X	vwmaccus	111110	V	F	vfwmsac
111111					111111	V	X	vwmaccsu	111111	V	F	vfwnmsac

Table 61. VRXUNARYO encoding space

vs2	
00000	vmv.s.x

Table 62. VWXUNARYO encoding space

vs1	
00000	vmv.x.s
10000	vcpop
10001	vfirst

Table 63. VXUNARYO encoding space

vs1	
00010	vzext.vf8
00011	vsext.vf8
00100	vzext.vf4
00101	vsext.vf4
00110	vzext.vf2
00111	vsext.vf2

Table 64. VRFUNARYO encoding space

vs2	
00000	vfmv.s.f

Table 65. VWFUNARYO encoding space

vs1	
00000	vfmv.f.s

Table 66. VFUNARYO encoding space

vs1	name
single-width converts	
00000	vfcvt.xu.f.v
00001	vfcvt.x.f.v
00010	vfcvt.f.xu.v
00011	vfcvt.f.x.v
00110	vfcvt.rtz.xu.f.v
00111	vfcvt.rtz.x.f.v
widening converts	
01000	vfwcvt.xu.f.v
01001	vfwcvt.x.f.v
01010	vfwcvt.f.xu.v
01011	vfwcvt.f.x.v
01100	vfwcvt.f.f.v

vs1	name
01110	vfwcvt.rtz.xu.f.v
01111	vfwcvt.rtz.x.f.v
narrowing converts	
10000	vfncvt.xu.f.w
10001	vfncvt.x.f.w
10010	vfncvt.f.xu.w
10011	vfncvt.f.x.w
10100	vfncvt.f.f.w
10101	vfncvt.rod.f.f.w
10110	vfncvt.rtz.xu.f.w
10111	vfncvt.rtz.x.f.w

Table 67. VFUNARY1 encoding space

vs1	name
00000	vfsqrt.v
00100	vfrsqrt7.v
00101	vfrec7.v
10000	vfclass.v

Table 68. VMUNARY0 encoding space

vs1	
00001	vmsbf
00010	vmsof
00011	vmsif
10000	viota
10001	vid

Chapter 32. Cryptography Extensions: Scalar & Entropy Source Instructions, Version 1.0.1

32.1. Introduction

This document describes the *scalar* cryptography extension for RISC-V. All instructions described herein use the general-purpose `X` registers, and obey the 2-read-1-write register access constraint. These instructions are designed to be lightweight and suitable for `32` and `64` bit base architectures; from embedded IoT class cores to large, application class cores which do not implement a vector unit.

This document also describes the architectural interface to an Entropy Source, which can be used to generate cryptographic secrets. This is found in Section 32.4.

It also contains a mechanism allowing core implementers to provide *"Constant Time Execution"* guarantees in Section 32.5.

32.1.1. Intended Audience

Cryptography is a specialised subject, requiring people with many different backgrounds to cooperate in its secure and efficient implementation. Where possible, we have written this specification to be understandable by all, though we recognise that the motivations and references to algorithms or other specifications and standards may be unfamiliar to those who are not domain experts.

This specification anticipates being read and acted on by various people with different backgrounds. We have tried to capture these backgrounds here, with a brief explanation of what we expect them to know, and how it relates to the specification. We hope this aids people's understanding of which aspects of the specification are particularly relevant to them, and which they may (safely!) ignore or pass to a colleague.

Cryptographers and cryptographic software developers

These are the people we expect to write code using the instructions in this specification. They should understand fairly obviously the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards to which we refer. We expect the sections on constant time execution (Section 32.5) and the entropy source (Section 32.4) to be chiefly understood with their help.

Computer architects

We do not expect architects to have a cryptography background. We nonetheless expect architects to be able to examine our instructions for implementation issues, understand how the instructions will be used in context, and advise on how best to fit the functionality the cryptographers want to the ISA interface.

Digital design engineers & micro-architects

These are the people who will implement the specification inside a core. Again, no cryptography expertise is assumed, but we expect them to interpret the specification and anticipate any hardware implementation issues, e.g., where high-frequency design considerations apply, or where latency/area tradeoffs exist etc. In particular, they should be aware of the literature around efficiently implementing AES and SM4 SBoxes in hardware.

Verification engineers

Responsible for ensuring the correct implementation of the extension in hardware. No cryptography background is assumed. We expect them to identify interesting test cases from the specification. An understanding of their real-world usage will help with this. We do not expect verification engineers in this sense to be experts in entropy source design or certification, since this is a very specialised area. We

do expect them however to identify all of the *architectural* test cases around the entropy source interface.

These are by no means the only people concerned with the specification, but they are the ones we considered most while writing it.

32.1.2. Sail Specifications

RISC-V maintains a formal model of the ISA specification, implemented in the Sail ISA specification language (*SAIL ISA Specification Language*, n.d.). Note that *Sail* refers to the specification language itself, and that there is a *model of RISC-V*, written using Sail. It is not correct to refer to "the Sail model". This is ambiguous, given there are many models of different ISAs implemented using Sail. We refer to the Sail implementation of RISC-V as "the RISC-V Sail model".

The Cryptography extension uses inline Sail code snippets from the actual model to give canonical descriptions of instruction functionality. Each instruction is accompanied by its expression in Sail, and includes calls to supporting functions which are too verbose to include directly in the specification. This supporting code is listed in Section 32.9. The Sail Manual is recommended reading in order to best understand the code snippets.

Note that this document contains only a subset of the formal model: refer to the formal model GitHub repository for the complete model.

32.1.3. Policies

In creating this proposal, we tried to adhere to the following policies:

- Where there is a choice between:
  1. supporting diverse implementation strategies for an algorithm or
  2. supporting a single implementation style which is more performant / less expensive; the crypto extension will pick the more constrained but performant option. This fits a common pattern in other parts of the RISC-V specification, where recommended (but not required) instruction sequences for performing particular tasks are given as an example, such that both hardware and software implementers can optimise for only a single use-case.
- The extension will be designed to support *existing* standardised cryptographic constructs well. It will not try to support proposed standards, or cryptographic constructs which exist only in academia. Cryptographic standards which are settled upon concurrently with or after the RISC-V cryptographic extension standardisation will be dealt with by future additions to, or versions of, the RISC-V cryptographic standard extension. It is anticipated that the NIST Lightweight Cryptography contest and the NIST Post-Quantum Cryptography contest may be dealt with this way, depending on timescales.
- Historically, there has been some discussion (Lee et al., 2004) on how newly supported operations in general-purpose computing might enable new bases for cryptographic algorithms. The standard will not try to anticipate new useful low-level operations which *may* be useful as building blocks for future cryptographic constructs.
- Regarding side-channel countermeasures: Where relevant, proposed instructions must aim to remove the possibility of any timing side-channels. For side-channels based on power or electro-magnetic (EM) measurements, the extension will not aim to support countermeasures which are implemented above the ISA abstraction layer. Recommendations will be given where relevant on how micro-architectures can implement instructions in a power/EM side-channel resistant way.

32.2. Extensions Overview

The group of extensions introduced by the Scalar Cryptography Instruction Set Extension is listed here.

Detection of individual cryptography extensions uses the unified software-based RISC-V discovery method.

At the time of writing, these discovery mechanisms are still a work in progress.

A note on extension rationale

Specialist encryption and decryption instructions are separated into different functional groups because some use cases (e.g., Galois/Counter Mode in TLS 1.3) do not require decryption functionality.

The NIST and ShangMi algorithms suites are separated because their usefulness is heavily dependent on the countries a device is expected to operate in. NIST ciphers are a part of most standardised internet protocols, while ShangMi ciphers are required for use in China.

32.2.1. `Zbkb` - Bitmanip instructions for Cryptography

This extension contains bit-manipulation instructions that are particularly useful for cryptography, most of which are also in the `Zbb` extension. Please refer to Bit-manipulation for Cryptography.

32.2.2. `Zbkc` - Carry-less multiply instructions

Constant time carry-less multiply for Galois/Counter Mode. These are separated from the Bit-manipulation for Cryptography because they have a considerable implementation overhead which cannot be amortised across other instructions.

Please refer to Carry-less multiplication for Cryptography.

32.2.3. `Zbkx` - Crossbar permutation instructions

These instructions are useful for implementing SBoxes in constant time, and potentially with DPA protections. These are separated from the Bit-manipulation for Cryptography because they have an implementation overhead which cannot be amortised across other instructions.

Please refer to Crossbar permutations.

32.2.4. `Zknd` - NIST Suite: AES Decryption

Instructions for accelerating the decryption and key-schedule functions of the AES block cipher.

RV32	RV64	Mnemonic	Instruction
✓		aes32dsi	AES final round decrypt (RV32)
✓		aes32dsmi	AES middle round decrypt (RV32)
	✓	aes64ds	AES decrypt final round (RV64)
	✓	aes64dsm	AES decrypt middle round (RV64)
	✓	aes64im	AES Decrypt KeySchedule MixColumns (RV64)
	✓	aes64ks1i	AES Key Schedule Instruction 1 (RV64)

RV32	RV64	Mnemonic	Instruction
	✓	aes64ks2	AES Key Schedule Instruction 2 (RV64)

The AES Key Schedule Instruction 1 (RV64) and AES Key Schedule Instruction 2 (RV64) instructions are present in both the Zknd and Zkne extensions.

32.2.5. `Zkne` - NIST Suite: AES Encryption

Instructions for accelerating the encryption and key-schedule functions of the AES block cipher.

RV32	RV64	Mnemonic	Instruction
✓		aes32esi	AES final round encrypt (RV32)
✓		aes32esmi	AES middle round encrypt (RV32)
	✓	aes64es	AES encrypt final round instruction (RV64)
	✓	aes64esm	AES encrypt middle round instruction (RV64)
	✓	aes64ks1i	AES Key Schedule Instruction 1 (RV64)
	✓	aes64ks2	AES Key Schedule Instruction 2 (RV64)

The `aes64ks1i` and `aes64ks2` instructions are present in both the Zknd and Zkne extensions.

32.2.6. `Zknh` - NIST Suite: Hash Function Instructions

Instructions for accelerating the SHA2 family of cryptographic hash functions, as specified in (NIST, 2015).

RV32	RV64	Mnemonic	Instruction
✓	✓	sha256sig0	SHA2-256 Sigma0 instruction
✓	✓	sha256sig1	SHA2-256 Sigma1 instruction
✓	✓	sha256sum0	SHA2-256 Sum0 instruction
✓	✓	sha256sum1	SHA2-256 Sum1 instruction
✓		sha512sig0h	SHA2-512 Sigma0 high (RV32)
✓		sha512sig0l	SHA2-512 Sigma0 low (RV32)
✓		sha512sig1h	SHA2-512 Sigma1 high (RV32)
✓		sha512sig1l	SHA2-512 Sigma1 low (RV32)
✓		sha512sum0r	SHA2-512 Sum0 (RV32)
✓		sha512sum1r	SHA2-512 Sum1 (RV32)
	✓	sha512sig0	SHA2-512 Sigma0 instruction (RV64)
	✓	sha512sig1	SHA2-512 Sigma1 instruction (RV64)
	✓	sha512sum0	SHA2-512 Sum0 instruction (RV64)
	✓	sha512sum1	SHA2-512 Sum1 instruction (RV64)

32.2.7. `Zksed` - ShangMi Suite: SM4 Block Cipher Instructions

Instructions for accelerating the SM4 Block Cipher. Note that unlike AES, this cipher uses the same core operation for encryption and decryption, hence there is only one extension for it.

RV32	RV64	Mnemonic	Instruction
✓	✓	sm4ed	SM4 Encrypt/Decrypt Instruction
✓	✓	sm4ks	SM4 Key Schedule Instruction

32.2.8. `Zksh` - ShangMi Suite: SM3 Hash Function Instructions

Instructions for accelerating the SM3 hash function.

RV32	RV64	Mnemonic	Instruction
✓	✓	sm3p0	SM3 P0 transform
✓	✓	sm3p1	SM3 P1 transform

32.2.9. `Zkr` - Entropy Source Extension

The entropy source extension defines the `seed` CSR at address `0x015`. This CSR provides up to 16 physical `entropy` bits that can be used to seed cryptographic random bit generators.

See Section 32.4 for the normative specification and access control notes. Section 32.7 contains design rationale and further recommendations to implementers.

32.2.10. `Zkn` - NIST Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zbkb	Bitmanipulation instructions for cryptography.
Zbkc	Carry-less multiply instructions.
Zbkx	Cross-bar Permutation instructions.
Zkne	AES encryption instructions.
Zknd	AES decryption instructions.
Zknh	SHA2 hash function instructions.

A core which implements `Zkn` must implement all of the above extensions.

32.2.11. `Zks` - ShangMi Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zbkb	Bitmanipulation instructions for cryptography.
Zbkc	Carry-less multiply instructions.
Zbkx	Cross-bar Permutation instructions.
Zksed	SM4 block cipher instructions.
Zksh	SM3 hash function instructions.

A core which implements `Zks` must implement all of the above extensions.

32.2.12. `Zk` - Standard scalar cryptography extension

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zkn	NIST Algorithm suite extension.
Zkr	Entropy Source extension.
Zkt	Data independent execution latency extension.

A core which implements `Zk` must implement all of the above extensions.

32.2.13. `Zkt` - Data Independent Execution Latency

This extension allows CPU implementers to indicate to cryptographic software developers that a subset of RISC-V instructions are guaranteed to be implemented such that their execution latency is independent of the data values they operate on. A complete description of this extension is found in Section 32.5.

32.3. Instructions

32.3.1. aes32dsi

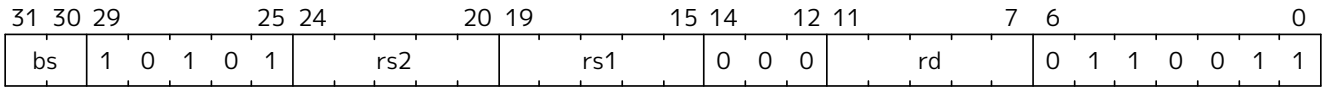
Synopsis

AES final round decryption instruction for RV32.

Mnemonic

aes32dsi rd, rs1, rs2, bs

Encoding



Description

This instruction sources a single byte from `rs2` according to `bs`. To this it applies the inverse AES SBox operation, and XOR's the result with `rs1`. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES32DSI (bs,rs2,rs1,rd)) = {
  let shamt   : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
  let si      : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
  let so      : bits(32) = 0x000000 @ aes_sbox_inv(si);
  let result  : bits(32) = X(rs1)[31..0] ^ rol32(so, unsigned(shamt));
  X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknd (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.2. aes32dsmi

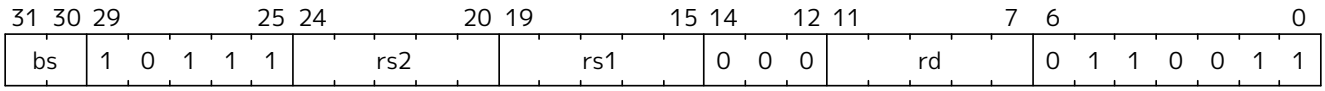
Synopsis

AES middle round decryption instruction for RV32.

Mnemonic

aes32dsmi rd, rs1, rs2, bs

Encoding



Description

This instruction sources a single byte from `rs2` according to `bs`. To this it applies the inverse AES SBox operation, and a partial inverse MixColumn, before XOR'ing the result with `rs1`. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES32DSMI (bs,rs2,rs1,rd)) = {
  let shamt   : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
  let si      : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
  let so      : bits( 8) = aes_sbox_inv(si);
  let mixed   : bits(32) = aes_mixcolumn_byte_inv(so);
  let result  : bits(32) = X(rs1)[31..0] ^ rol32(mixed, unsigned(shamt));
  X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknd (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.3. aes32esi

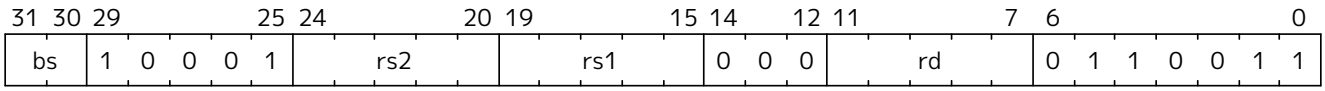
Synopsis

AES final round encryption instruction for RV32.

Mnemonic

aes32esi rd, rs1, rs2, bs

Encoding



Description

This instruction sources a single byte from `rs2` according to `bs`. To this it applies the forward AES SBox operation, before XOR'ing the result with `rs1`. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES32ESI (bs,rs2,rs1,rd)) = {
  let shamt   : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
  let si      : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
  let so      : bits(32) = 0x000000 @ aes_sbox_fwd(si);
  let result  : bits(32) = X(rs1)[31..0] ^ rol32(so, unsigned(shamt));
  X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.4. aes32esmi

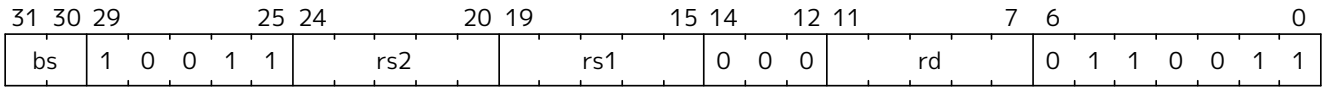
Synopsis

AES middle round encryption instruction for RV32.

Mnemonic

aes32esmi rd, rs1, rs2, bs

Encoding



Description

This instruction sources a single byte from `rs2` according to `bs`. To this it applies the forward AES SBox operation, and a partial forward MixColumn, before XOR'ing the result with `rs1`. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES32ESMI (bs,rs2,rs1,rd)) = {
  let shamt   : bits( 5) = bs @ 0b000; /* shamt = bs*8 */
  let si      : bits( 8) = (X(rs2)[31..0] >> shamt)[7..0]; /* SBox Input */
  let so      : bits( 8) = aes_sbox_fwd(si);
  let mixed   : bits(32) = aes_mixcolumn_byte_fwd(so);
  let result  : bits(32) = X(rs1)[31..0] ^ rol32(mixed, unsigned(shamt));
  X(rd) = EXTS(result); RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.5. aes64ds

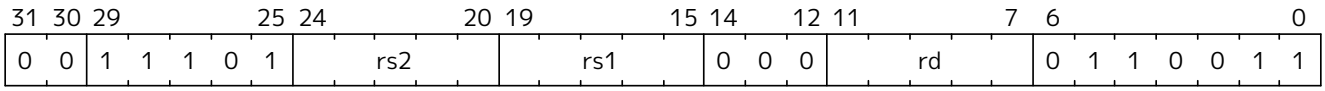
Synopsis

AES final round decryption instruction for RV64.

Mnemonic

aes64ds rd, rs1, rs2

Encoding



Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the Inverse ShiftRows and SubBytes steps. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note To Software Developers

The following code snippet shows the final round of the AES block decryption. `t0` *and* `t1` *hold the current round state.* `t2` *and* `t3` *hold the next round state.*

```
aes64ds t2, t0, t1
aes64ds t3, t1, t0
```

Note the reversed register order of the second instruction.

Operation

```
function clause execute (AES64DS(rs2, rs1, rd)) = {
  let sr : bits(64) = aes_rv64_shiftrows_inv(X(rs2)[63..0], X(rs1)[63..0]);
  let wd : bits(64) = sr[63..0];
  X(rd) = aes_apply_inv_sbox_to_each_byte(wd);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknd (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.6. aes64dsm

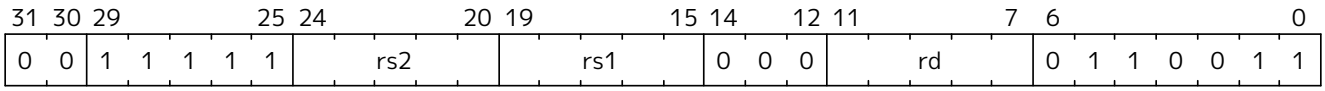
Synopsis

AES middle round decryption instruction for RV64.

Mnemonic

aes64dsm rd, rs1, rs2

Encoding



Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the Inverse ShiftRows, SubBytes and MixColumns steps. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note To Software Developers

The following code snippet shows one middle round of the AES block decryption. `t0` *and* `t1` *hold the current round state.* `t2` *and* `t3` *hold the next round state.*

```
aes64dsm t2, t0, t1
aes64dsm t3, t1, t0
```

Note the reversed register order of the second instruction.

Operation

```
function clause execute (AES64DSM(rs2, rs1, rd)) = {
  let sr : bits(64) = aes_rv64_shiftrows_inv(X(rs2)[63..0], X(rs1)[63..0]);
  let wd : bits(64) = sr[63..0];
  let sb : bits(64) = aes_apply_inv_sbox_to_each_byte(wd);
  X(rd)  = aes_mixcolumn_inv(sb[63..32]) @ aes_mixcolumn_inv(sb[31..0]);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknd (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.7. aes64es

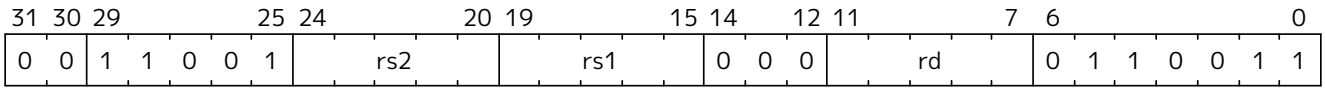
Synopsis

AES final round encryption instruction for RV64.

Mnemonic

aes64es rd, rs1, rs2

Encoding



Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the ShiftRows and SubBytes steps. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note To Software Developers

The following code snippet shows the final round of the AES block encryption. `t0` *and* `t1` *hold the current round state.* `t2` *and* `t3` *hold the next round state.*

```
aes64es t2, t0, t1
aes64es t3, t1, t0
```

Note the reversed register order of the second instruction.

Operation

```
function clause execute (AES64ES(rs2, rs1, rd)) = {
  let sr : bits(64) = aes_rv64_shiftrows_fwd(X(rs2)[63..0], X(rs1)[63..0]);
  let wd : bits(64) = sr[63..0];
  X(rd) = aes_apply_fwd_sbox_to_each_byte(wd);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.8. aes64esm

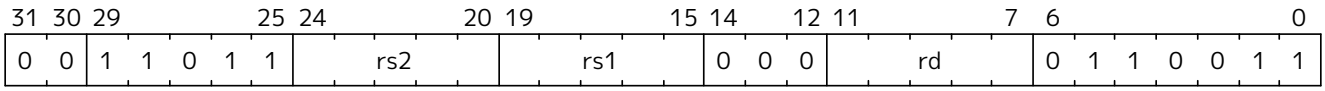
Synopsis

AES middle round encryption instruction for RV64.

Mnemonic

aes64esm rd, rs1, rs2

Encoding



Description

Uses the two 64-bit source registers to represent the entire AES state, and produces *half* of the next round output, applying the ShiftRows, SubBytes and MixColumns steps. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note To Software Developers

The following code snippet shows one middle round of the AES block encryption. `t0` *and* `t1` *hold the current round state.* `t2` *and* `t3` *hold the next round state.*

```
aes64esm t2, t0, t1
aes64esm t3, t1, t0
```

Note the reversed register order of the second instruction.

Operation

```
function clause execute (AES64ESM(rs2, rs1, rd)) = {
  let sr : bits(64) = aes_rv64_shiftrows_fwd(X(rs2)[63..0], X(rs1)[63..0]);
  let wd : bits(64) = sr[63..0];
  let sb : bits(64) = aes_apply_fwd_sbox_to_each_byte(wd);
  X(rd)  =  aes_mixcolumn_fwd(sb[63..32]) @ aes_mixcolumn_fwd(sb[31..0]);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.9. aes64im

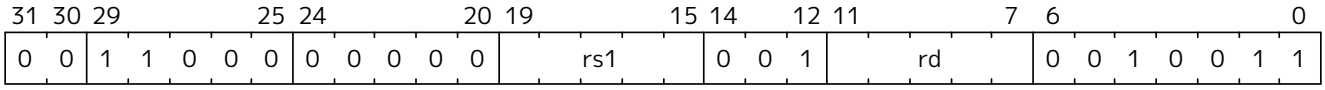
Synopsis

This instruction accelerates the inverse MixColumns step of the AES Block Cipher, and is used to aid creation of the decryption KeySchedule.

Mnemonic

aes64im rd, rs1

Encoding



Description

The instruction applies the inverse MixColumns transformation to two columns of the state array, packed into a single 64-bit register. It is used to create the inverse cipher KeySchedule, according to the equivalent inverse cipher construction in (NIST, 2001) (Page 23, Section 5.3.5). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES64IM(rs1, rd)) = {
  let w0 : bits(32) = aes_mixcolumn_inv(X(rs1)[31.. 0]);
  let w1 : bits(32) = aes_mixcolumn_inv(X(rs1)[63..32]);
  X(rd)  = w1 @ w0;
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknd (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.10. aes64ks1i

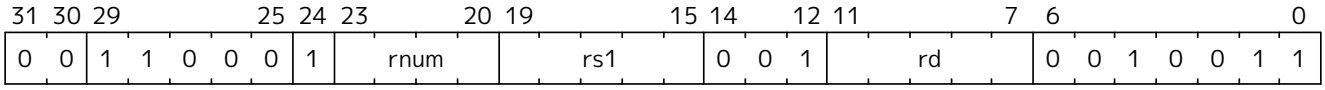
Synopsis

This instruction implements part of the KeySchedule operation for the AES Block cipher involving the SBox operation.

Mnemonic

aes64ks1i rd, rs1, rnum

Encoding



Description

This instruction implements the rotation, SubBytes and Round Constant addition steps of the AES block cipher Key Schedule. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on. Note that `rnum` must be in the range `0x0..0xA`. The values `0xB..0xF` are reserved.

Operation

```
function clause execute (AES64KS1I(rnum, rs1, rd)) = {
  if(unsigned(rnum) > 10) then {
    handle_illegal();  RETIRE_SUCCESS
  } else {
    let tmp1 : bits(32) = X(rs1)[63..32];
    let rc   : bits(32) = aes_decode_rcon(rnum); /* round number -> round
constant */
    let tmp2 : bits(32) = if (rnum ==0xA) then tmp1 else ror32(tmp1, 8);
    let tmp3 : bits(32) = aes_subword_fwd(tmp2);
    let result : bits(64) = (tmp3 ^ rc) @ (tmp3 ^ rc);
    X(rd) = EXTZ(result);
    RETIRE_SUCCESS
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV64)	v1.0.0	Ratified
Zknd (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.11. aes64ks2

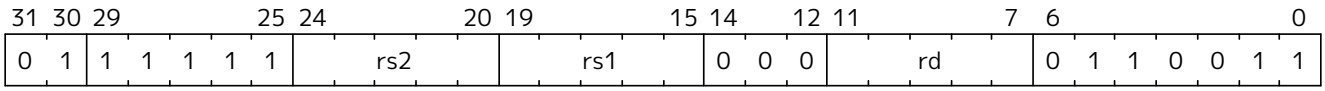
Synopsis

This instruction implements part of the KeySchedule operation for the AES Block cipher.

Mnemonic

aes64ks2 rd, rs1, rs2

Encoding



Description

This instruction implements the additional XOR'ing of key words as part of the AES block cipher Key Schedule. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (AES64KS2(rs2, rs1, rd)) = {
  let w0 : bits(32) = X(rs1)[63..32] ^ X(rs2)[31..0];
  let w1 : bits(32) = X(rs1)[63..32] ^ X(rs2)[31..0] ^ X(rs2)[63..32];
  X(rd)  = w1 @ w0;
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zkne (RV64)	v1.0.0	Ratified
Zknd (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.12. andn

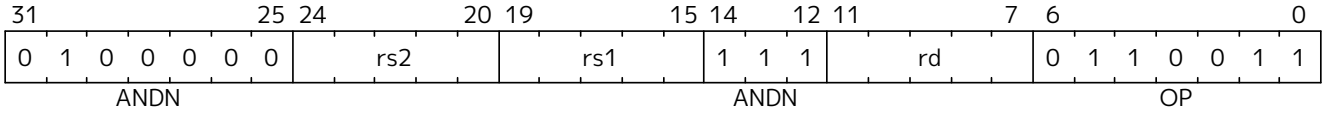
Synopsis

AND with inverted operand

Mnemonic

andn *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bitwise logical AND operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) & ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.13. brev8

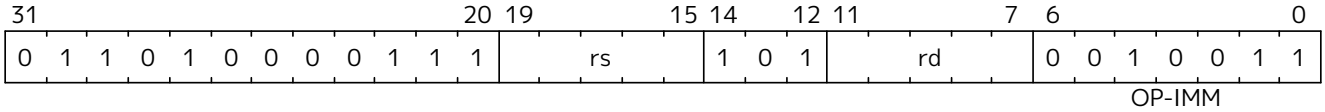
Synopsis

Reverse the bits in each byte of a source register.

Mnemonic

brev8 *rd*, *rs*

Encoding



Description

This instruction reverses the order of the bits in every byte of a register.

Operation

```
result : xlenbits = EXTZ(0b0);
foreach (i from 0 to sizeof(xlen) by 8) {
    result[i+7..i] = reverse_bits_in_byte(X(rs1)[i+7..i]);
};
X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.14. clmul

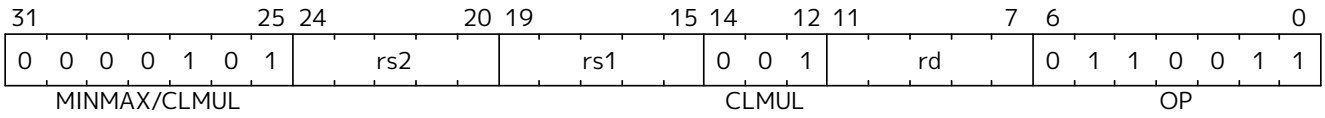
Synopsis

Carry-less multiply (low-part)

Mnemonic

clmul *rd*, *rs1*, *rs2*

Encoding



Description

clmul produces the lower half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 0 to (xlen - 1) by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val << i);
            else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	1.0.0	Ratified
Zbkc (Zbkc)	v1.0.0-rc4	Ratified

32.3.15. clmulh

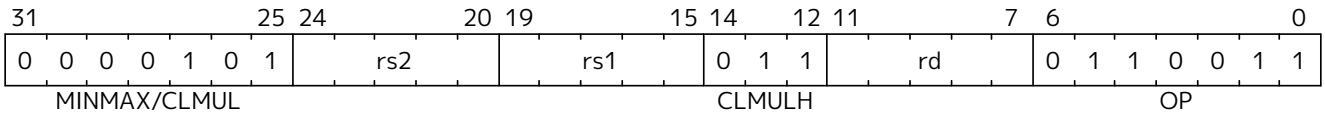
Synopsis

Carry-less multiply (high-part)

Mnemonic

clmulh *rd*, *rs1*, *rs2*

Encoding



Description

clmulh produces the upper half of the 2·XLEN carry-less product.

Operation

```
let rs1_val = X(rs1);
let rs2_val = X(rs2);
let output : xlenbits = 0;

foreach (i from 1 to xlen by 1) {
   output = if   ((rs2_val >> i) & 1)
            then output ^ (rs1_val >> (xlen - i));
            else output;
}

X[rd] = output
```

Included in

Extension	Minimum version	Lifecycle state
Zbc (Carry-less multiplication)	1.0.0	Ratified
Zbkc (Zbkc)	v1.0.0-rc4	Ratified

32.3.16. orn

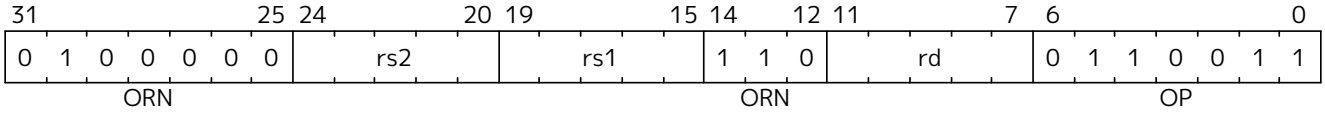
Synopsis

OR with inverted operand

Mnemonic

orn *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bitwise logical OR operation between *rs1* and the bitwise inversion of *rs2*.

Operation

```
X(rd) = X(rs1) | ~X(rs2);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.17. pack

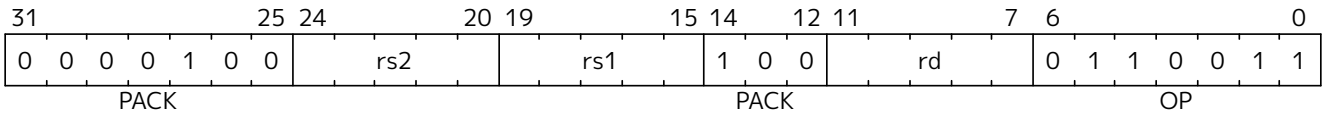
Synopsis

Pack the low halves of *rs1* and *rs2* into *rd*.

Mnemonic

pack *rd*, *rs1*, *rs2*

Encoding



Description

The pack instruction packs the XLEN/2-bit lower halves of *rs1* and *rs2* into *rd*, with *rs1* in the lower half and *rs2* in the upper half.

Operation

```
let lo_half : bits(xlen/2) = X(rs1)[xlen/2-1..0];
let hi_half : bits(xlen/2) = X(rs2)[xlen/2-1..0];
X(rd) = EXTZ(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.18. packh

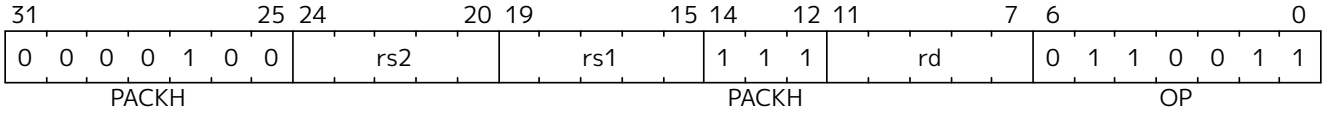
Synopsis

Pack the low bytes of *rs1* and *rs2* into *rd*.

Mnemonic

packh *rd*, *rs1*, *rs2*

Encoding



Description

And the packh instruction packs the least-significant bytes of *rs1* and *rs2* into the 16 least-significant bits of *rd*, zero extending the rest of *rd*.

Operation

```
let lo_half : bits(8) = X(rs1)[7..0];
let hi_half : bits(8) = X(rs2)[7..0];
X(rd) = EXTZ(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.19. packw

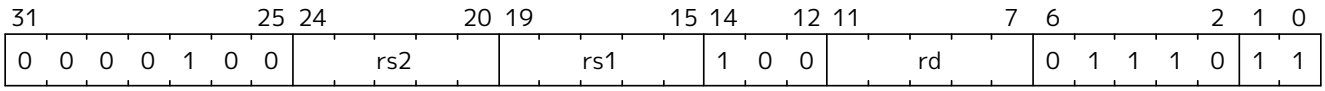
Synopsis

Pack the low 16-bits of *rs1* and *rs2* into *rd* on RV64.

Mnemonic

packw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction packs the low 16 bits of *rs1* and *rs2* into the 32 least-significant bits of *rd*, sign extending the 32-bit result to the rest of *rd*. This instruction only exists on RV64 based systems.

Operation

```
let lo_half : bits(16) = X(rs1)[15..0];
let hi_half : bits(16) = X(rs2)[15..0];
X(rd) = EXTS(hi_half @ lo_half);
```

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.20. rev8

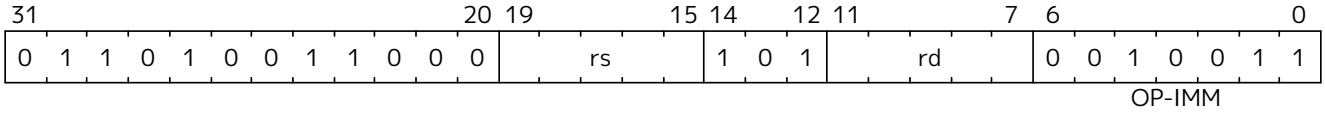
Synopsis

Byte-reverse register

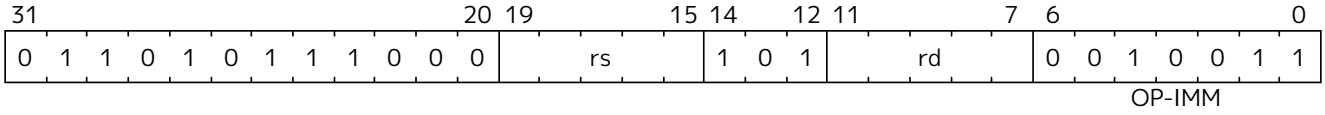
Mnemonic

rev8 *rd*, *rs*

Encoding (RV32)



Encoding (RV64)



Description

This instruction reverses the order of the bytes in *rs*.

Operation

```
let input = X(rs);
let output : xlenbits = 0;
let j = xlen - 1;

foreach (i from 0 to (xlen - 8) by 8) {
   output[i..(i + 7)] = input[(j - 7)..j];
   j = j - 8;
}

X[rd] = output
```

Note

The **rev8** *mnemonic corresponds to different instruction encodings in RV32 and RV64.*

Software Hint

The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 rd,rs` *followed by a* `srai rd,rd,K`*, where K is XLEN-32 and XLEN-16, respectively.*

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.21. rol

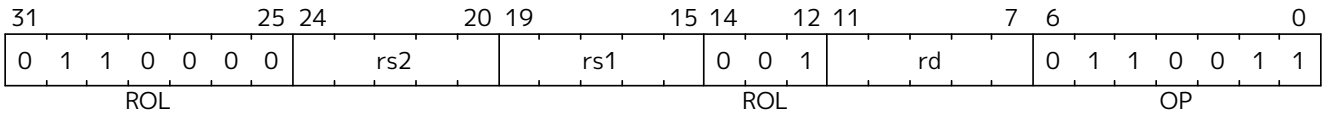
Synopsis

Rotate Left (Register)

Mnemonic

rol *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
            else X(rs2)[5..0];
let result = (X(rs1) << shamt) | (X(rs1) >> (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.22. rolw

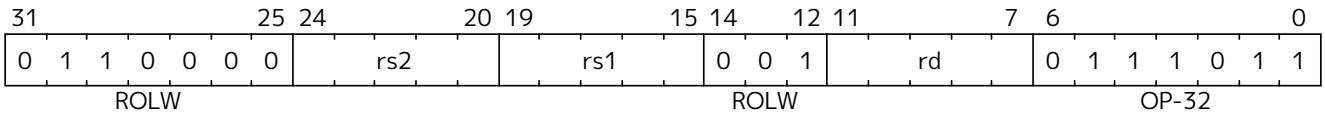
Synopsis

Rotate Left Word (Register)

Mnemonic

rolw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 << shamt) | (rs1 >> (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.23. ror

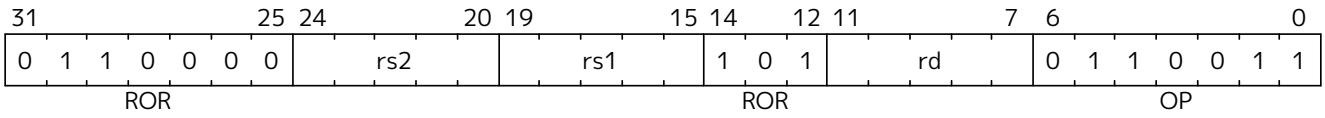
Synopsis

Rotate Right

Mnemonic

ror *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right of *rs1* by the amount in least-significant log2(XLEN) bits of *rs2*.

Operation

```
let shamt = if   xlen == 32
            then X(rs2)[4..0]
            else X(rs2)[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.24. rori

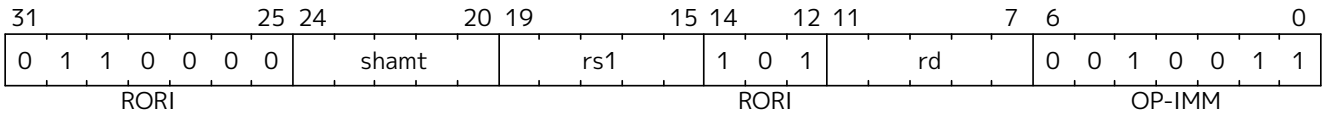
Synopsis

Rotate Right (Immediate)

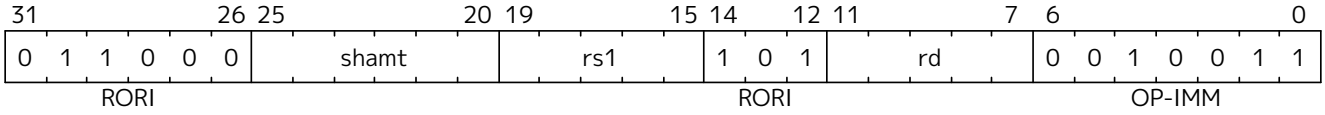
Mnemonic

rori *rd*, *rs1*, *shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction performs a rotate right of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

Operation

```
let shamt = if   xlen == 32
            then shamt[4..0]
            else shamt[5..0];
let result = (X(rs1) >> shamt) | (X(rs1) << (xlen - shamt));

X(rd) = result;
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.25. roriw

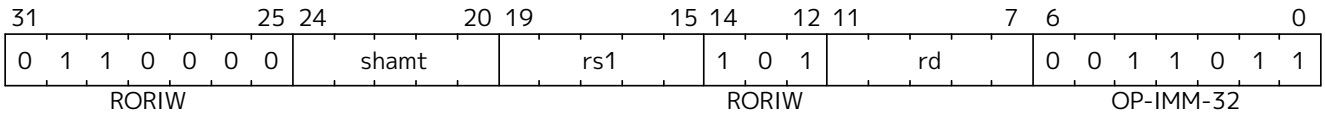
Synopsis

Rotate Right Word by Immediate

Mnemonic

roriw *rd*, *rs1*, *shamt*

Encoding



Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in the least-significant log2(XLEN) bits of *shamt*. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1_data = EXTZ(X(rs1)[31..0];
let result = (rs1_data >> shamt) | (rs1_data << (32 - shamt));
X(rd) = EXTS(result[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.26. rorw

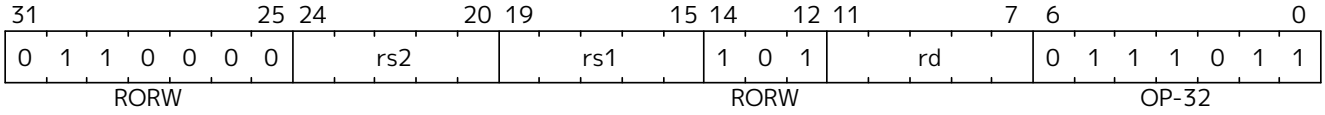
Synopsis

Rotate Right Word (Register)

Mnemonic

rorw *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right on the least-significant word of *rs1* by the amount in least-significant 5 bits of *rs2*. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

Operation

```
let rs1 = EXTZ(X(rs1)[31..0])
let shamt = X(rs2)[4..0];
let result = (rs1 >> shamt) | (rs1 << (32 - shamt));
X(rd) = EXTS(result);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.27. sha256sig0

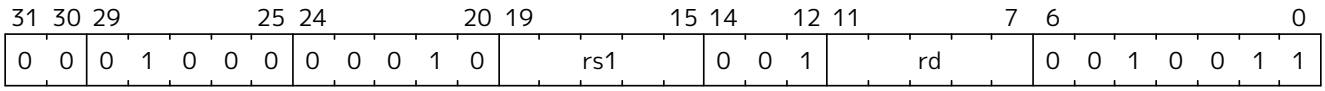
Synopsis

Implements the Sigma0 transformation function as used in the SHA2-256 hash function (NIST, 2015).

Mnemonic

sha256sig0 rd, rs1

Encoding



Description

This instruction is supported for both RV32 and RV64 base architectures. For RV32, the entire `XLEN` source register is operated on. For RV64, the low `32` bits of the source register are operated on, and the result sign extended to `XLEN` bits. Though named for SHA2-256, the instruction works for both the SHA2-224 and SHA2-256 parameterizations as described in (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA256SIG0(rs1,rd)) = {
  let inb    : bits(32) = X(rs1)[31..0];
  let result : bits(32) = ror32(inb,  7) ^ ror32(inb, 18) ^ (inb >>  3);
  X(rd)      = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh	v1.0.0	Ratified
Zkn	v1.0.0	Ratified
Zk	v1.0.0	Ratified

32.3.28. sha256sig1

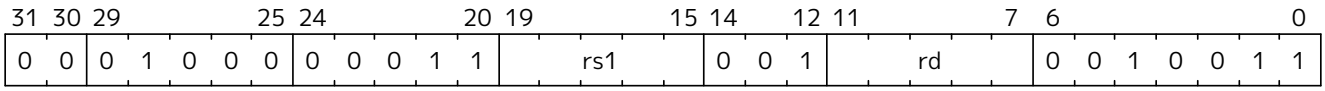
Synopsis

Implements the Sigma1 transformation function as used in the SHA2-256 hash function (NIST, 2015).

Mnemonic

sha256sig1 rd, rs1

Encoding



Description

This instruction is supported for both RV32 and RV64 base architectures. For RV32, the entire `XLEN` source register is operated on. For RV64, the low `32` bits of the source register are operated on, and the result sign extended to `XLEN` bits. Though named for SHA2-256, the instruction works for both the SHA2-224 and SHA2-256 parameterizations as described in (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA256SIG1(rs1,rd)) = {
  let inb    : bits(32) = X(rs1)[31..0];
  let result : bits(32) = ror32(inb, 17) ^ ror32(inb, 19) ^ (inb >> 10);
  X(rd)      = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh	v1.0.0	Ratified
Zkn	v1.0.0	Ratified
Zk	v1.0.0	Ratified

32.3.29. sha256sum0

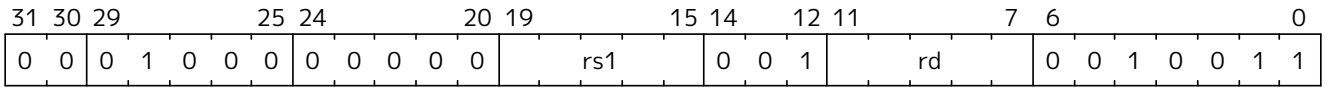
Synopsis

Implements the Sum0 transformation function as used in the SHA2-256 hash function (NIST, 2015).

Mnemonic

sha256sum0 rd, rs1

Encoding



Description

This instruction is supported for both RV32 and RV64 base architectures. For RV32, the entire `XLEN` source register is operated on. For RV64, the low `32` bits of the source register are operated on, and the result sign extended to `XLEN` bits. Though named for SHA2-256, the instruction works for both the SHA2-224 and SHA2-256 parameterizations as described in (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA256SUM0(rs1,rd)) = {
  let inb    : bits(32) = X(rs1)[31..0];
  let result : bits(32) = ror32(inb,  2) ^ ror32(inb, 13) ^ ror32(inb, 22);
  X(rd)      = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh	v1.0.0	Ratified
Zkn	v1.0.0	Ratified
Zk	v1.0.0	Ratified

32.3.30. sha256sum1

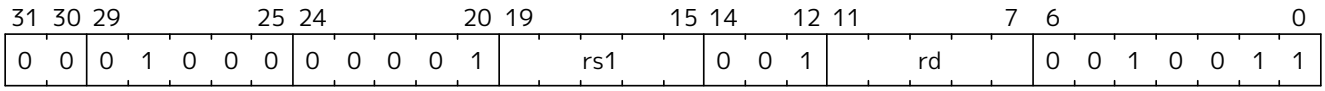
Synopsis

Implements the Sum1 transformation function as used in the SHA2-256 hash function (NIST, 2015).

Mnemonic

sha256sum1 rd, rs1

Encoding



Description

This instruction is supported for both RV32 and RV64 base architectures. For RV32, the entire `XLEN` source register is operated on. For RV64, the low `32` bits of the source register are operated on, and the result sign extended to `XLEN` bits. Though named for SHA2-256, the instruction works for both the SHA2-224 and SHA2-256 parameterizations as described in (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA256SUM1(rs1,rd)) = {
  let inb    : bits(32) = X(rs1)[31..0];
  let result : bits(32) = ror32(inb,  6) ^ ror32(inb, 11) ^ ror32(inb, 25);
  X(rd)      = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh	v1.0.0	Ratified
Zkn	v1.0.0	Ratified
Zk	v1.0.0	Ratified

32.3.31. sha512sig0h

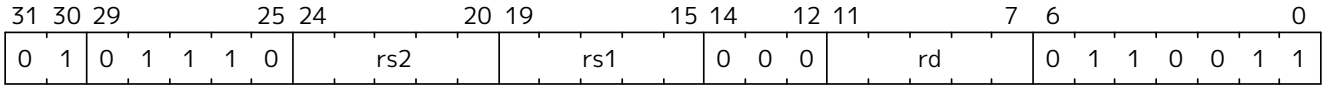
Synopsis

Implements the *high half* of the Sigma0 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig0h rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sigma0 transform of the SHA2-512 hash function in conjunction with the `sha512sig0l` instruction. The transform is a 64-bit to 64-bit function, so the input and output are each represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sigma0 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sig0l    t0, a0, a1
sha512sig0h    t1, a1, a0
```

Operation

```
function clause execute (SHA512SIG0H(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) >>  1) ^ (X(rs1) >>  7) ^ (X(rs1) >>  8) ^
               (X(rs2) << 31)                  ^ (X(rs2) << 24) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.32. sha512sig0l

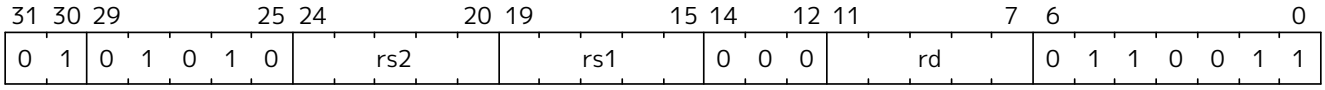
Synopsis

Implements the *low half* of the Sigma0 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig0l rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sigma0 transform of the SHA2-512 hash function in conjunction with the `sha512sig0h` instruction. The transform is a 64-bit to 64-bit function, so the input and output are each represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sigma0 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sig0l    t0, a0, a1
sha512sig0h    t1, a1, a0
```

Operation

```
function clause execute (SHA512SIG0L(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) >>  1) ^ (X(rs1) >>  7) ^ (X(rs1) >>  8) ^
               (X(rs2) << 31) ^ (X(rs2) << 25) ^ (X(rs2) << 24) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.33. sha512sig1h

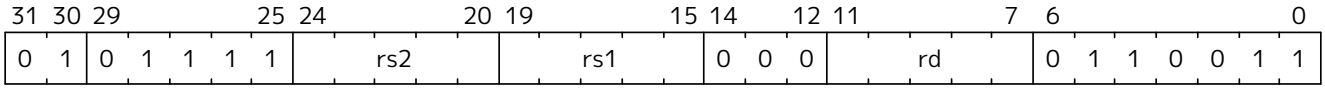
Synopsis

Implements the *high half* of the Sigma1 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig1h rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sigma1 transform of the SHA2-512 hash function in conjunction with the `sha512sig1l` instruction. The transform is a 64-bit to 64-bit function, so the input and output are each represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sigma1 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sig1l    t0, a0, a1
sha512sig1h    t1, a1, a0
```

Operation

```
function clause execute (SHA512SIG1H(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) <<  3) ^ (X(rs1) >>  6) ^ (X(rs1) >> 19) ^
               (X(rs2) >> 29)                  ^ (X(rs2) << 13) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.34. sha512sig1l

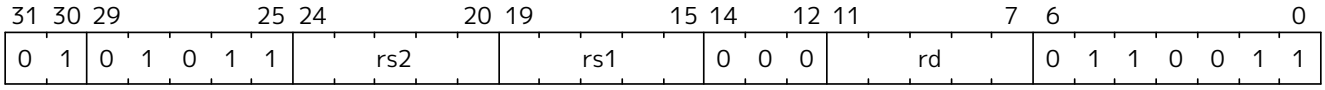
Synopsis

Implements the *low half* of the Sigma1 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig1l rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sigma1 transform of the SHA2-512 hash function in conjunction with the `sha512sig1h` instruction. The transform is a 64-bit to 64-bit function, so the input and output are each represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sigma1 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sig1l    t0, a0, a1
sha512sig1h    t1, a1, a0
```

Operation

```
function clause execute (SHA512SIG1L(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) <<  3) ^ (X(rs1) >>  6) ^ (X(rs1) >> 19) ^
               (X(rs2) >> 29) ^ (X(rs2) << 26) ^ (X(rs2) << 13) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.35. sha512sum0r

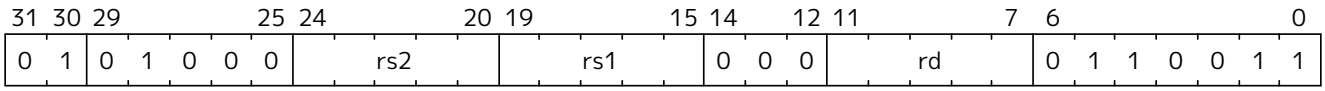
Synopsis

Implements the Sum0 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sum0r rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sum0 transform of the SHA2-512 hash function. The transform is a 64-bit to 64-bit function, so the input and output is represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sum0 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sum0r    t0, a0, a1
sha512sum0r    t1, a1, a0
```

Note the reversed source register ordering.

Operation

```
function clause execute (SHA512SUM0R(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) << 25) ^ (X(rs1) << 30) ^ (X(rs1) >> 28) ^
               (X(rs2) >>  7) ^ (X(rs2) >>  2) ^ (X(rs2) <<  4) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.36. sha512sum1r

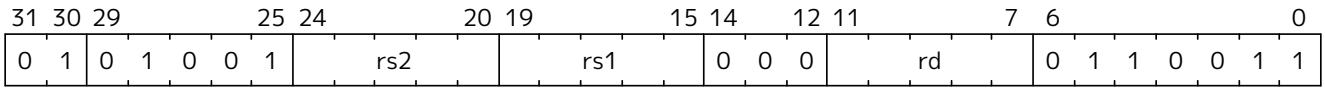
Synopsis

Implements the Sum1 transformation, as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sum1r rd, rs1, rs2

Encoding



Description

This instruction is implemented on RV32 only. Used to compute the Sum1 transform of the SHA2-512 hash function. The transform is a 64-bit to 64-bit function, so the input and output is represented by two 32-bit registers. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Note to software developers

The entire Sum1 transform for SHA2-512 may be computed on RV32 using the following instruction sequence:

```
sha512sum1r    t0, a0, a1
sha512sum1r    t1, a1, a0
```

Note the reversed source register ordering.

Operation

```
function clause execute (SHA512SUM1R(rs2, rs1, rd)) = {
  X(rd) = EXTS((X(rs1) << 23) ^ (X(rs1) >> 14) ^ (X(rs1) >> 18) ^
               (X(rs2) >>  9) ^ (X(rs2) << 18) ^ (X(rs2) << 14) );
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV32)	v1.0.0	Ratified
Zkn (RV32)	v1.0.0	Ratified
Zk (RV32)	v1.0.0	Ratified

32.3.37. sha512sig0

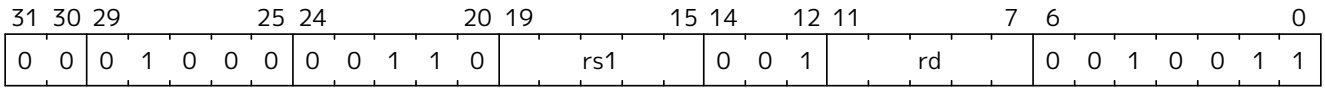
Synopsis

Implements the Sigma0 transformation function as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig0 rd, rs1

Encoding



Description

This instruction is supported for the RV64 base architecture. It implements the Sigma0 transform of the SHA2-512 hash function. (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA512SIG0(rs1, rd)) = {
  X(rd) = ror64(X(rs1),  1) ^ ror64(X(rs1),  8) ^ (X(rs1) >> 7);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.38. sha512sig1

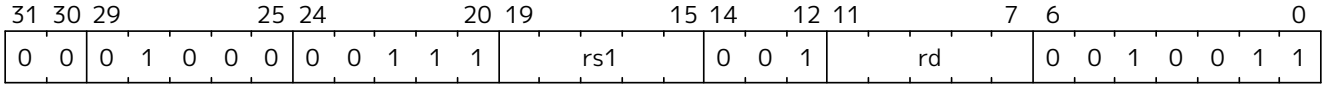
Synopsis

Implements the Sigma1 transformation function as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sig1 rd, rs1

Encoding



Description

This instruction is supported for the RV64 base architecture. It implements the Sigma1 transform of the SHA2-512 hash function. (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA512SIG1(rs1, rd)) = {
  X(rd) = ror64(X(rs1), 19) ^ ror64(X(rs1), 61) ^ (X(rs1) >> 6);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.39. sha512sum0

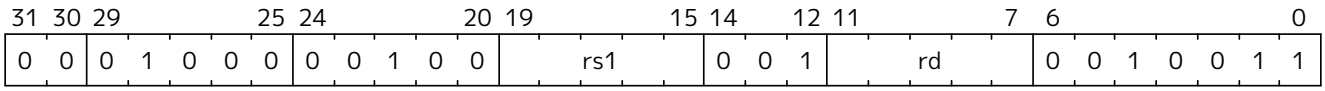
Synopsis

Implements the Sum0 transformation function as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sum0 rd, rs1

Encoding



Description

This instruction is supported for the RV64 base architecture. It implements the Sum0 transform of the SHA2-512 hash function. (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA512SUM0(rs1, rd)) = {
  X(rd) = ror64(X(rs1), 28) ^ ror64(X(rs1), 34) ^ ror64(X(rs1) ,39);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.40. sha512sum1

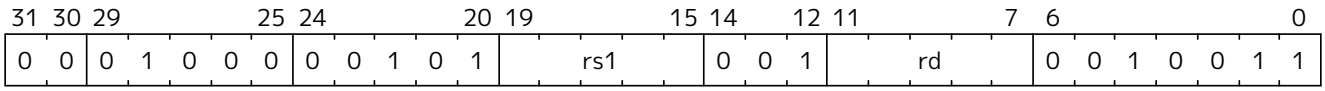
Synopsis

Implements the Sum1 transformation function as used in the SHA2-512 hash function (NIST, 2015).

Mnemonic

sha512sum1 rd, rs1

Encoding



Description

This instruction is supported for the RV64 base architecture. It implements the Sum1 transform of the SHA2-512 hash function. (NIST, 2015). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SHA512SUM1(rs1, rd)) = {
  X(rd) = ror64(X(rs1), 14) ^ ror64(X(rs1), 18) ^ ror64(X(rs1) ,41);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zknh (RV64)	v1.0.0	Ratified
Zkn (RV64)	v1.0.0	Ratified
Zk (RV64)	v1.0.0	Ratified

32.3.41. sm3p0

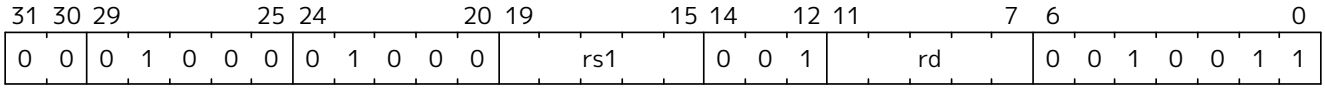
Synopsis

Implements the *P0* transformation function as used in the SM3 hash function (*GB/T 32905-2016: SM3 Cryptographic Hash Algorithm*, 2016; ISO/IEC, 2018).

Mnemonic

sm3p0 rd, rs1

Encoding



Description

This instruction is supported for the RV32 and RV64 base architectures. It implements the *P0* transform of the SM3 hash function (*GB/T 32905-2016: SM3 Cryptographic Hash Algorithm*, 2016; ISO/IEC, 2018). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Supporting Material

This instruction is based on work done in (Saarinen, 2020).

Operation

```
function clause execute (SM3P0(rs1, rd)) = {
  let r1     : bits(32) = X(rs1)[31..0];
  let result : bits(32) =  r1 ^ rol32(r1,  9) ^ rol32(r1, 17);
  X(rd) = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zksh	v1.0.0	Ratified
Zks	v1.0.0	Ratified

32.3.42. sm3p1

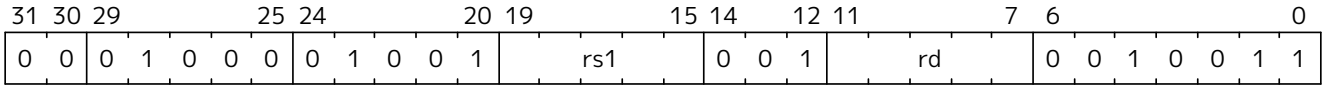
Synopsis

Implements the *P1* transformation function as used in the SM3 hash function (*GB/T 32905-2016: SM3 Cryptographic Hash Algorithm*, 2016; ISO/IEC, 2018).

Mnemonic

sm3p1 rd, rs1

Encoding



Description

This instruction is supported for the RV32 and RV64 base architectures. It implements the *P1* transform of the SM3 hash function (*GB/T 32905-2016: SM3 Cryptographic Hash Algorithm*, 2016; ISO/IEC, 2018). This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Supporting Material

This instruction is based on work done in (Saarinen, 2020).

Operation

```
function clause execute (SM3P1(rs1, rd)) = {
  let r1     : bits(32) = X(rs1)[31..0];
  let result : bits(32) =  r1 ^ rol32(r1, 15) ^ rol32(r1, 23);
  X(rd) = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zksh	v1.0.0	Ratified
Zks	v1.0.0	Ratified

32.3.43. sm4ed

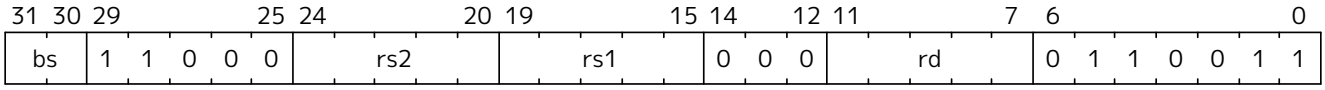
Synopsis

Accelerates the block encrypt/decrypt operation of the SM4 block cipher (*GB/T 32907-2016: SM4 Block Cipher Algorithm*, 2016; ISO/IEC, 2018).

Mnemonic

sm4ed rd, rs1, rs2, bs

Encoding



Description

Implements a T-tables in hardware style approach to accelerating the SM4 round function. A byte is extracted from `rs2` based on `bs`, to which the SBox and linear layer transforms are applied, before the result is XOR'd with `rs1` and written back to `rd`. This instruction exists on RV32 and RV64 base architectures. On RV64, the 32-bit result is sign extended to XLEN bits. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SM4ED (bs,rs2,rs1,rd)) = {
  let shamt : bits(5)  = bs @ 0b000; /* shamt = bs*8 */
  let sb_in : bits(8)  = (X(rs2)[31..0] >> shamt)[7..0];
  let x     : bits(32) = 0x000000 @ sm4_sbox(sb_in);
  let y     : bits(32) = x ^ (x               <<  8) ^ ( x               <<  2)
^
                         (x               << 18) ^ ((x & 0x0000003F) << 26)
^
                         ((x & 0x000000C0) << 10);
  let z     : bits(32) = rol32(y, unsigned(shamt));
  let result: bits(32) = z ^ X(rs1)[31..0];
  X(rd)                = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zksed	v1.0.0	Ratified
Zks	v1.0.0	Ratified

32.3.44. sm4ks

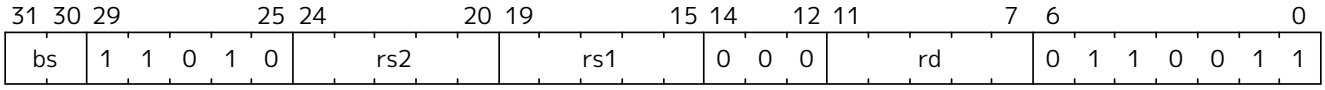
Synopsis

Accelerates the Key Schedule operation of the SM4 block cipher (*GB/T 32907-2016: SM4 Block Cipher Algorithm*, 2016; ISO/IEC, 2018).

Mnemonic

sm4ks rd, rs1, rs2, bs

Encoding



Description

Implements a T-tables in hardware style approach to accelerating the SM4 Key Schedule. A byte is extracted from `rs2` based on `bs`, to which the SBox and linear layer transforms are applied, before the result is XOR'd with `rs1` and written back to `rd`. This instruction exists on RV32 and RV64 base architectures. On RV64, the 32-bit result is sign extended to XLEN bits. This instruction must *always* be implemented such that its execution latency does not depend on the data being operated on.

Operation

```
function clause execute (SM4KS (bs,rs2,rs1,rd)) = {
  let shamt : bits(5)  = (bs @ 0b000); /* shamt = bs*8 */
  let sb_in : bits(8)  = (X(rs2)[31..0] >> shamt)[7..0];
  let x     : bits(32) = 0x000000 @ sm4_sbox(sb_in);
  let y     : bits(32) = x ^ ((x & 0x00000007) << 29) ^ ((x & 0x000000FE) <<
7) ^
                         ((x & 0x00000001) << 23) ^ ((x & 0x000000F8) <<
13) ;
  let z     : bits(32) = rol32(y, unsigned(shamt));
  let result: bits(32) = z ^ X(rs1)[31..0];
  X(rd) = EXTS(result);
  RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zksed	v1.0.0	Ratified
Zks	v1.0.0	Ratified

32.3.45. unzip

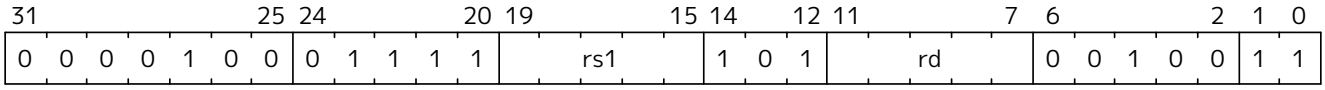
Synopsis

Place odd and even bits of the source register into upper and lower halves of the destination register, respectively.

Mnemonic

unzip *rd*, *rs*

Encoding



Description

This instruction scatters all of the odd and even bits of a source word into the high and low halves of a destination word. It is the inverse of the zip instruction. This instruction is available only on RV32.

Operation

```
foreach (i from 0 to xlen/2-1) {
  X(rd)[i] = X(rs1)[2*i]
  X(rd)[i+xlen/2] = X(rs1)[2*i+1]
}
```

Software Hint

This instruction is useful for implementing the SHA3 cryptographic hash function on a 32-bit architecture, as it implements the bit-interleaving operation used to speed up the 64-bit rotations directly.

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb) (RV32)	v1.0.0-rc4	Ratified

32.3.46. xnor

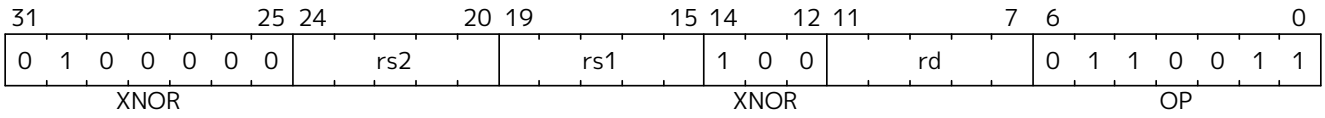
Synopsis

Exclusive NOR

Mnemonic

xnor *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs the bit-wise exclusive-NOR operation on *rs1* and *rs2*.

Operation

```
X(rd) = ~(X(rs1) ^ X(rs2));
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

32.3.47. xperm8

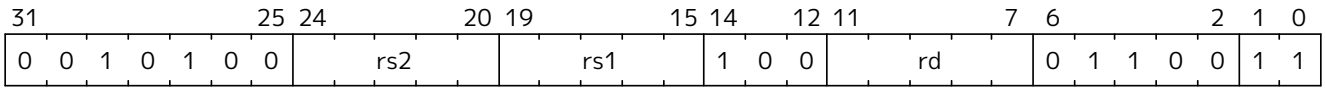
Synopsis

Byte-wise lookup of indices into a vector in registers.

Mnemonic

xperm8 *rd*, *rs1*, *rs2*

Encoding



Description

The xperm8 instruction operates on bytes. The *rs1* register contains a vector of XLEN/8 8-bit elements. The *rs2* register contains a vector of XLEN/8 8-bit indexes. The result is each element in *rs2* replaced by the indexed element in *rs1*, or zero if the index into *rs2* is out of bounds.

Operation

```
val xperm8_lookup : (bits(8), xlenbits) -> bits(8)
function xperm8_lookup (idx, lut) = {
    (lut >> (idx @ 0b000))[7..0]
}

function clause execute ( XPERM8 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 8) {
        result[i+7..i] = xperm8_lookup(X(rs2)[i+7..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zbkx (Crossbar permutations)	v1.0	Ratified

32.3.48. xperm4

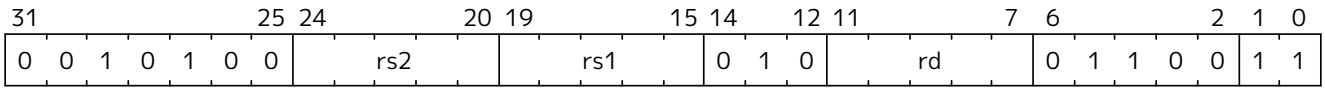
Synopsis

Nibble-wise lookup of indices into a vector.

Mnemonic

xperm4 *rd*, *rs1*, *rs2*

Encoding



Description

The xperm4 instruction operates on nibbles. The *rs1* register contains a vector of XLEN/4 4-bit elements. The *rs2* register contains a vector of XLEN/4 4-bit indexes. The result is each element in *rs2* replaced by the indexed element in *rs1*, or zero if the index into *rs2* is out of bounds.

Operation

```
val xperm4_lookup : (bits(4), xlenbits) -> bits(4)
function xperm4_lookup (idx, lut) = {
    (lut >> (idx @ 0b00))[3..0]
}

function clause execute ( XPERM4 (rs2,rs1,rd)) = {
    result : xlenbits = EXTZ(0b0);
    foreach(i from 0 to xlen by 4) {
        result[i+3..i] = xperm4_lookup(X(rs2)[i+3..i], X(rs1));
    };
    X(rd) = result;
    RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Zbkx (Crossbar permutations)	v1.0	Ratified

32.3.49. zip

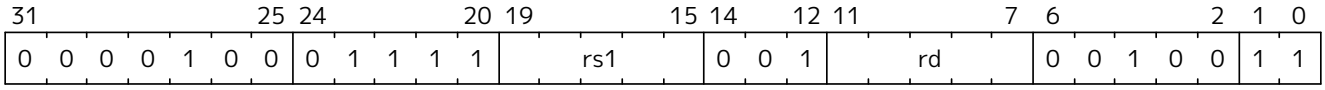
Synopsis

Interleave upper and lower halves of the source register into odd and even bits of the destination register, respectively.

Mnemonic

zip *rd*, *rs*

Encoding



Description

This instruction gathers bits from the high and low halves of the source word into odd/even bit positions in the destination word. It is the inverse of the unzip instruction. This instruction is available only on RV32.

Operation

```
foreach (i from 0 to xlen/2-1) {
  X(rd)[2*i] = X(rs1)[i]
  X(rd)[2*i+1] = X(rs1)[i+xlen/2]
}
```

Software Hint

This instruction is useful for implementing the SHA3 cryptographic hash function on a 32-bit architecture, as it implements the bit-interleaving operation used to speed up the 64-bit rotations directly.

Included in

Extension	Minimum version	Lifecycle state
Zbkb (Zbkb) (RV32)	v1.0.0-rc4	Ratified

32.4. Entropy Source

The `seed` CSR provides an interface to a NIST SP 800-90B (Turan et al., 2018) or BSI AIS-31 (Killmann & Schindler, 2011) compliant physical Entropy Source (ES).

An entropy source, by itself, is not a cryptographically secure Random Bit Generator (RBG), but can be used to build standard (and nonstandard) RBGs of many types with the help of symmetric cryptography. Expected usage is to condition (typically with SHA-2/3) the output from an entropy source and use it to seed a cryptographically secure Deterministic Random Bit Generator (DRBG) such as AES-based `CTR_DRBG` (Barker & Kelsey, 2015). The combination of an Entropy Source, Conditioning, and a DRBG can be used to create random bits securely (Barker et al., 2021). See Section 32.7 for a non-normative description of a certification and self-certification procedures, design rationale, and more detailed suggestions on how the entropy source output can be used.

32.4.1. The `seed` CSR

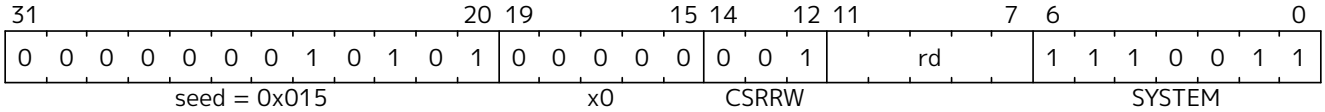
`seed` is an unprivileged CSR located at address `0x015`. The 32-bit contents of `seed` are as follows:

Bits	Name	Description
`31:30`	`OPST`	Status: `BIST` (00), `WAIT` (01), `ES16` (10), `DEAD` (11).
`29:24`	*reserved*	For future use by the RISC-V specification.
`23:16`	*custom*	Designated for custom and experimental use.
`15: 0`	`entropy`	16 bits of randomness, only when `OPST=ES16`.

Attempts to access the `seed` CSR using a read-only CSR-access instruction (`CSRRS`/`CSRRC` with *rs1*=`x0` or `CSRRSI`/`CSRRCI` with *uimm*=0) raise an illegal-instruction exception; any other CSR-access instruction may be used to access `seed`. The write value (in `rs1` or `uimm`) must be ignored by implementations. The purpose of the write is to signal polling and flushing.

Software normally uses the instruction `csrrw rd, seed, x0` to read the `seed` CSR.

Encoding



The `seed` CSR is also access controlled by execution mode, and attempted read or write access will raise an illegal-instruction exception outside M mode unless access is explicitly granted. See Section 32.4.3 for more details.

The status bits `seed[31:30]` = `OPST` may be `ES16` (10), indicating successful polling, or one of three entropy polling failure statuses `BIST` (00), `WAIT` (01), or `DEAD` (11), discussed below.

Each returned `seed[15:0]` = `entropy` value represents unique randomness when `OPST`=`ES16` (`seed[31:30]` = `10`), even if its numerical value is the same as that of a previously polled `entropy` value. The implementation requirements of `entropy` bits are defined in Section 32.4.2. When `OPST` is not `ES16`, `entropy` must be set to 0. An implementation may safely set reserved and custom bits to zeros.

For security reasons, the interface guarantees that secret `entropy` words are not made available multiple times. Hence polling (reading) must also have the side effect of clearing (wipe-on-read) the `entropy` contents and changing the state to `WAIT` (unless there is `entropy` immediately available for `ES16`). Other states (`BIST`, `WAIT`, and `DEAD`) may be unaffected by polling.

The Status Bits returned in `seed[31:30]`=`OPST`:

- `00` - `BIST` indicates that Built-In Self-Test "on-demand" (BIST) testing is being performed. If `OPST` returns temporarily to `BIST` from any other state, this signals a non-fatal self-test alarm, which is non-actionable, apart from being logged. Such a `BIST` alarm must be latched until polled at least once to enable software to record its occurrence.
- `01` - `WAIT` means that a sufficient amount of entropy is not yet available. This is not an error condition and may (in fact) be more frequent than ES16 since physical entropy sources often have low bandwidth.
- `10` - `ES16` indicates success; the low bits `seed[15:0]` will have 16 bits of randomness (`entropy`), which is guaranteed to meet certain minimum entropy requirements, regardless of implementation.
- `11` - `DEAD` is an unrecoverable self-test error. This may indicate a hardware fault, a security issue, or (extremely rarely) a type-1 statistical false positive in the continuous testing procedures. In case of a fatal failure, an immediate lockdown may also be an appropriate response in dedicated security devices.

Example. `0x8000ABCD` is a valid `ES16` status output, with `0xABCD` being the `entropy` value. `0xFFFFFFFF` is an invalid output (`DEAD`) with no `entropy` value.

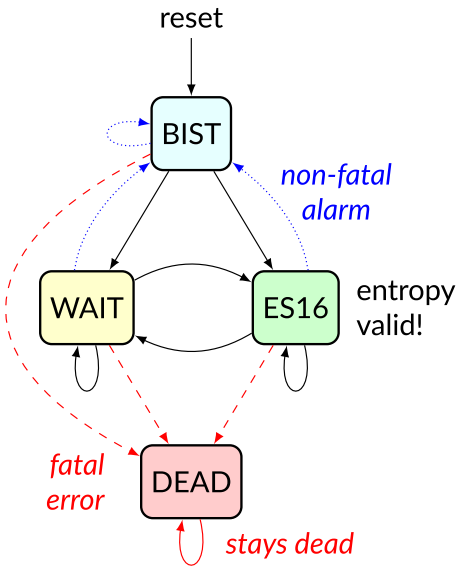


Figure 6. Entropy Source state transition diagram.

Normally the operational state alternates between WAIT (no data) and ES16, which means that 16 bits of randomness (`entropy`) have been polled. BIST (Built-in Self-Test) only occurs after reset or to signal a non-fatal self-test alarm (if reached after WAIT or ES16). DEAD is an unrecoverable error state.

32.4.2. Entropy Source Requirements

The output `entropy` (`seed[15:0]` in ES16 state) is not necessarily fully conditioned randomness due to hardware and energy limitations of smaller, low-powered implementations. However, minimum requirements are defined. The main requirement is that 2-to-1 cryptographic post-processing in 256-bit input blocks will yield 128-bit "full entropy" output blocks. Entropy source users may make this conservative assumption but are not prohibited from using more than twice the number of seed bits relative to the desired resulting entropy.

An implementation of the entropy source should meet at least one of the following requirements sets in

order to be considered a secure and safe design:

- Section 32.4.2.1: A physical entropy source meeting NIST SP 800-90B (Turan et al., 2018) criteria with evaluated min-entropy of 192 bits for each 256 output bits (min-entropy rate 0.75).
- Section 32.4.2.2: A physical entropy source meeting the AIS-31 PTG.2 (Killmann & Schindler, 2011) criteria, implying average Shannon entropy rate 0.997. The source must also meet the NIST 800-90B min-entropy rate 192/256 = 0.75.
- Section 32.4.2.3: A virtual entropy source is a DRBG seeded from a physical entropy source. It must have at least a 256-bit (Post-Quantum Category 5) internal security level.

All implementations must signal initialization, test mode, and health alarms as required by respective standards. This may require the implementer to add non-standard (custom) test interfaces in a secure and safe manner, an example of which is described in Section 32.7.6

32.4.2.1. NIST SP 800-90B / FIPS 140-3 Requirements

All NIST SP 800-90B (Turan et al., 2018) required components and health test mechanisms must be implemented.

The entropy requirement is satisfied if 128 bits of *full entropy* can be obtained from each 256-bit (16*16 -bit) successful, but possibly non-consecutive `entropy` (ES16) output sequence using a vetted conditioning algorithm such as a cryptographic hash (See Section 3.1.5.1.1, SP 800-90B (Turan et al., 2018)). In practice, a min-entropy rate of 0.75 or larger is required for this.

Note that 128 bits of estimated input min-entropy does not yield 128 bits of conditioned, full entropy in SP 800-90B/C evaluation. Instead, the implication is that every 256-bit sequence should have min-entropy of at least 128+64 = 192 bits, as discussed in SP 800-90C (Barker et al., 2021); the likelihood of successfully "guessing" an individual 256-bit output sequence should not be higher than 2^{-192} even with (almost) unconstrained amount of entropy source data and computational power.

Rather than attempting to define all the mathematical and architectural properties that the entropy source must satisfy, we define that the physical entropy source be strong and robust enough to pass the equivalent of NIST SP 800-90 evaluation and certification for full entropy when conditioned cryptographically in ratio 2:1 with 128-bit output blocks.

Even though the requirement is defined in terms of 128-bit full entropy blocks, we recommend 256-bit security. This can be accomplished by using at least 512 `entropy` bits to initialize a DRBG that has 256-bit security.

32.4.2.2. BSI AIS-31 PTG.2 / Common Criteria Requirements

For alternative Common Criteria certification (or self-certification), AIS 31 PTG.2 class (Killmann & Schindler, 2011) (Sect. 4.3.) required hardware components and mechanisms must be implemented. In addition to AIS-31 PTG.2 randomness requirements (Shannon entropy rate of 0.997 as evaluated in that standard), the overall min-entropy requirement of remains, as discussed in Section 32.4.2.1. Note that 800-90B min-entropy can be significantly lower than AIS-31 Shannon entropy. These two metrics should not be equated or confused with each other.

32.4.2.3. Virtual Sources: Security Requirement

A virtual source is not an ISA compliance requirement. It is defined for the benefit of the RISC-V security ecosystem so that virtual systems may have a consistent level of security.

A virtual source is not a physical entropy source but provides additional protection against covert channels, depletion attacks, and host identification in operating environments that can not be entirely trusted with direct access to a hardware resource. Despite limited trust, implementers should try to guarantee that even such environments have sufficient entropy available for secure cryptographic operations.

A virtual source traps access to the `seed` CSR, emulates it, or otherwise implements it, possibly without direct access to a physical entropy source. The output can be cryptographically secure pseudorandomness instead of real entropy, but must have at least 256-bit security, as defined below. A virtual source is intended especially for guest operating systems, sandboxes, emulators, and similar use cases.

As a technical definition, a random-distinguishing attack against the output should require computational resources comparable or greater than those required for exhaustive key search on a secure block cipher with a 256-bit key (e.g., AES 256). This applies to both classical and quantum computing models, but only classical information flows. The virtual source security requirement maps to Post-Quantum Security Category 5 (NIST, 2016).

Any implementation of the `seed` CSR that limits the security strength shall not reduce it to less than 256 bits. If the security level is under 256 bits, then the interface must not be available.

A virtual entropy source does not need to implement `WAIT` or `BIST` states. It should fail (`DEAD`) if the host DRBG or entropy source fails and there is insufficient seeding material for the host DRBG.

32.4.3. Access Control to `seed`

The Zkr extension adds the `SSEED` and `USEED` fields to the `mseccfg` CSR to control access to the `seed` CSR from U, S, or HS modes (see Privileged ISA specification).

Systems should implement carefully considered access control policies from lower privilege modes to physical entropy sources. The system can trap attempted access to `seed` and feed a less privileged client *virtual entropy source* data (Section 32.4.2.3) instead of invoking an SP 800-90B (Section 32.4.2.1) or PTG.2 (Section 32.4.2.2) *physical entropy source*. Emulated `seed` data generation is made with an appropriately seeded, secure software DRBG. See Section 32.7.3.5 for security considerations related to direct access to entropy sources.

Implementations may implement `mseccfg` such that `[s,u]seed` is a read-only constant value `0`. Software may discover if access to the `seed` CSR can be enabled in U and S mode by writing a `1` to `[s,u]seed` and reading back the result.

32.5. Data Independent Execution Latency Subset: Zkt

The Zkt extension attests that the machine has data-independent execution time for a safe subset of instructions. This property is commonly called *"constant-time"* although should not be taken with that literal meaning.

All currently defined cryptographic instructions (Zk* and Zbk* extensions) are on this list, together with a set of relevant supporting instructions from I, M, C, and B extensions.

Note to software developers

Failure to prevent leakage of sensitive parameters via the direct timing channel is considered a serious security vulnerability and will typically result in a CERT CVE security advisory.

32.5.1. Scope and Goal

An "ISA contract" is made between a programmer and the RISC-V implementation that Zkt instructions do not leak information about processed secret data (plaintext, keying information, or other "sensitive security parameters" — FIPS 140-3 term) through differences in execution latency. Zkt does *not* define a set of instructions available in the core; it just restricts the behaviour of certain instructions if those are implemented.

Currently, the scope of this document is within scalar RV32/RV64 processors. Vector cryptography instructions (and appropriate vector support instructions) will be added later, as will other security-related functions that wish to assert leakage-free execution latency properties.

Loads, stores, conditional branches are excluded, along with a set of instructions that are rarely necessary to process secret data. Also excluded are instructions for which workarounds exist in standard cryptographic middleware due to the limitations of other ISA processors.

The stated goal is that OpenSSL, BoringSSL (Android), the Linux Kernel, and similar trusted software will not have directly observable timing side channels when compiled and running on a Zkt-enabled RISC-V target. The Zkt extension explicitly states many of the common latency assumptions made by cryptography developers.

Vendors do not have to implement all of the list's instructions to be Zkt compliant; however, if they claim to have Zkt and implement any of the listed instructions, it must have data-independent latency.

For example, many simple RV32I and RV64I cores (without Multiply, Compressed, Bitmanip, or Cryptographic extensions) are technically compliant with Zkt. A constant-time AES can be implemented on them using "bit-slice" techniques, but it will be excruciatingly slow when compared to implementation with AES instructions. There are no guarantees that even a bit-sliced cipher implementation (largely based on boolean logic instructions) is secure on a core without Zkt attestation.

Out-of-order implementations adhering to Zkt are still free to fuse, crack, change or even ignore sequences of instructions, so long as the optimisations are applied deterministically, and not based on operand data. The guiding principle should be that no information about the data being operated on should be leaked based on the execution latency.

It is left to future extensions or other techniques to tackle the problem of data-independent execution in implementations which advanced out-of-order capabilities which use value prediction, or which are otherwise data-dependent.

Note to software developers

Programming techniques can only mitigate leakage directly caused by arithmetic, caches, and branches. Other ISAs have had micro-architectural issues such as Spectre, Meltdown, Speculative Store Bypass, Rogue System Register Read, Lazy FP State Restore, Bounds Check Bypass Store, TLBleed, and L1TF/Foreshadow, etc. See e.g. NSA Hardware and Firmware Security Guidance

It is not within the remit of this proposal to mitigate these micro-architectural leakages.

32.5.2. Background

- Timing attacks are much more powerful than was realised before the 2010s, which has led to a significant mitigation effort in current cryptographic code-bases.
- Cryptography developers use static and dynamic security testing tools to trace the handling of secret information and detect occasions where it influences a branch or is used for a table lookup.

- Architectural testing for Zkt can be pragmatic and semi-formal; *security by design* against basic timing attacks can usually be achieved via conscious implementation (of relevant iterative multi-cycle instructions or instructions composed of micro-ops) in way that avoids data-dependent latency.
- Laboratory testing may utilize statistical timing attack leakage analysis techniques such as those described in ISO/IEC 17825 (ISO, 2016).
- Binary executables should not contain secrets in the instruction encodings (Kerckhoffs's principle), so instruction timing may leak information about immediates, ordering of input registers, etc. There may be an exception to this in systems where a binary loader modifies the executable for purposes of relocation — and it is desirable to keep the execution location (PC) secret. This is why instructions such as LUI, AUIPC, and ADDI are on the list.
- The rules used by audit tools are relatively simple to understand. Very briefly; we call the plaintext, secret keys, expanded keys, nonces, and other such variables "secrets". A secret variable (arithmetically) modifying any other variable/register turns that into a secret too. If a secret ends up in address calculation affecting a load or store, that is a violation. If a secret affects a branch's condition, that is also a violation. A secret variable location or register becomes a non-secret via specific zeroization/sanitisation or by being declared ciphertext (or otherwise no-longer-secret information). In essence, secrets can only "touch" instructions on the Zkt list while they are secrets.

32.5.3. Specific Instruction Rationale

- HINT instruction forms (typically encodings with *rd*=`x0`) are excluded from the data-independent time requirement.
- Floating point (F, D, Q, L extensions) are currently excluded from the constant-time requirement as they have very few applications in standardised cryptography. We may consider adding floating point add, sub, multiply as a constant time requirement for some floating point extension in case a specific algorithm (such as the PQC Signature algorithm Falcon) becomes critical.
- Cryptographers typically assume division to be variable-time (while multiplication is constant time) and implement their Montgomery reduction routines with that assumption.
- Zicsr, Zifencei are excluded.
- Some instructions are on the list simply because we see no harm in including them in testing scope.

32.5.4. Programming Information

For background information on secure programming "models", see:

- Thomas Pornin: *"Why Constant-Time Crypto?"* (A great introduction to timing assumptions.) www.bearssl.org/constanttime.html
- Jean-Philippe Aumasson: *"Guidelines for low-level cryptography software."* (A list of recommendations.) github.com/veorq/cryptocoding
- Peter Schwabe: *"Timing Attacks and Countermeasures."* (Lecture slides — nice references.) summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf
- Adam Langley: *"ctgrind."* (This is from 2010 but is still relevant.) www.imperialviolet.org/2010/04/01/ctgrind.html
- Kris Kwiatkowski: *"Constant-time code verification with Memory Sanitizer."* www.amongbytes.com/post/20210709-testing-constant-time/
- For early examples of timing attack vulnerabilities, see www.kb.cert.org/vuls/id/997481 and related academic papers.

32.5.5. Zkt listings

The following instructions are included in the `Zkt` subset They are listed here grouped by their original parent extension.

Note to implementers

You do not need to implement all of these instructions to implement `Zkt`*. Rather, every one of these instructions that the core does implement must adhere to the requirements of* `Zkt`*.*

32.5.5.1. RVI (Base Instruction Set)

Only basic arithmetic and `slt*` (for carry computations) are included. The data-independent timing requirement does not apply to HINT instruction encoding forms of these instructions.

RV32	RV64	Mnemonic
✓	✓	lui *rd*, *imm*
✓	✓	auipc *rd*, *imm*
✓	✓	addi *rd*, *rs1*, *imm*
✓	✓	slti *rd*, *rs1*, *imm*
✓	✓	sltiu *rd*, *rs1*, *imm*
✓	✓	xori *rd*, *rs1*, *imm*
✓	✓	ori *rd*, *rs1*, *imm*
✓	✓	andi *rd*, *rs1*, *imm*
✓	✓	slli *rd*, *rs1*, *imm*
✓	✓	srli *rd*, *rs1*, *imm*
✓	✓	srai *rd*, *rs1*, *imm*
✓	✓	add *rd*, *rs1*, *rs2*
✓	✓	sub *rd*, *rs1*, *rs2*
✓	✓	sll *rd*, *rs1*, *rs2*
✓	✓	slt *rd*, *rs1*, *rs2*
✓	✓	sltu *rd*, *rs1*, *rs2*
✓	✓	xor *rd*, *rs1*, *rs2*
✓	✓	srl *rd*, *rs1*, *rs2*
✓	✓	sra *rd*, *rs1*, *rs2*
✓	✓	or *rd*, *rs1*, *rs2*
✓	✓	and *rd*, *rs1*, *rs2*
	✓	addiw *rd*, *rs1*, *imm*
	✓	slliw *rd*, *rs1*, *imm*
	✓	srliw *rd*, *rs1*, *imm*
	✓	sraiw *rd*, *rs1*, *imm*
	✓	addw *rd*, *rs1*, *rs2*
	✓	subw *rd*, *rs1*, *rs2*
	✓	sllw *rd*, *rs1*, *rs2*
	✓	srlw *rd*, *rs1*, *rs2*

RV32	RV64	Mnemonic
	✓	sraw *rd*, *rs1*, *rs2*

32.5.5.2. RVM (Multiply)

Multiplication is included; division and remaindering excluded.

RV32	RV64	Mnemonic
✓	✓	mul *rd*, *rs1*, *rs2*
✓	✓	mulh *rd*, *rs1*, *rs2*
✓	✓	mulhsu *rd*, *rs1*, *rs2*
✓	✓	mulhu *rd*, *rs1*, *rs2*
	✓	mulw *rd*, *rs1*, *rs2*

32.5.5.3. RVC (Compressed)

Same criteria as in RVI. Organised by quadrants.

RV32	RV64	Mnemonic
✓	✓	c.nop
✓	✓	c.addi
	✓	c.addiw
✓	✓	c.lui
✓	✓	c.srli
✓	✓	c.srai
✓	✓	c.andi
✓	✓	c.sub
✓	✓	c.xor
✓	✓	c.or
✓	✓	c.and
	✓	c.subw
	✓	c.addw
✓	✓	c.slli
✓	✓	c.mv
✓	✓	c.add

32.5.5.4. Zcb Extension

These instructions are compressed versions of I and M instructions that are included in Zkt.

RV32	RV64	Mnemonic	Instruction
✓	✓	c.mul	Multiply, 16-bit encoding
✓	✓	c.not	Bitwise not, 16-bit encoding

RV32	RV64	Mnemonic	Instruction
✓	✓	c.zext.b	Zero extend byte, 16-bit encoding

32.5.5.5. RVK (Scalar Cryptography)

All K-specific instructions are included. Additionally, `seed` CSR latency should be independent of `ES16` state output `entropy` bits, as that is a sensitive security parameter. See Section 32.7.3.5.

RV32	RV64	Mnemonic	Instruction
✓		aes32dsi	AES final round decrypt (RV32)
✓		aes32dsmi	AES middle round decrypt (RV32)
✓		aes32esi	AES final round encrypt (RV32)
✓		aes32esmi	AES middle round encrypt (RV32)
	✓	aes64ds	AES decrypt final round (RV64)
	✓	aes64dsm	AES decrypt middle round (RV64)
	✓	aes64es	AES encrypt final round instruction (RV64)
	✓	aes64esm	AES encrypt middle round instruction (RV64)
	✓	aes64im	AES Decrypt KeySchedule MixColumns (RV64)
	✓	aes64ks1i	AES Key Schedule Instruction 1 (RV64)
	✓	aes64ks2	AES Key Schedule Instruction 2 (RV64)
✓	✓	sha256sig0	SHA2-256 Sigma0 instruction
✓	✓	sha256sig1	SHA2-256 Sigma1 instruction
✓	✓	sha256sum0	SHA2-256 Sum0 instruction
✓	✓	sha256sum1	SHA2-256 Sum1 instruction
✓		sha512sig0h	SHA2-512 Sigma0 high (RV32)
✓		sha512sig0l	SHA2-512 Sigma0 low (RV32)
✓		sha512sig1h	SHA2-512 Sigma1 high (RV32)
✓		sha512sig1l	SHA2-512 Sigma1 low (RV32)
✓		sha512sum0r	SHA2-512 Sum0 (RV32)
✓		sha512sum1r	SHA2-512 Sum1 (RV32)
	✓	sha512sig0	SHA2-512 Sigma0 instruction (RV64)
	✓	sha512sig1	SHA2-512 Sigma1 instruction (RV64)
	✓	sha512sum0	SHA2-512 Sum0 instruction (RV64)
	✓	sha512sum1	SHA2-512 Sum1 instruction (RV64)
✓	✓	sm3p0	SM3 P0 transform
✓	✓	sm3p1	SM3 P1 transform
✓	✓	sm4ed	SM4 Encrypt/Decrypt Instruction
✓	✓	sm4ks	SM4 Key Schedule Instruction

32.5.5.6. RVB (Bitmanip)

The Zbkb, Zbkc and Zbkx extensions are included in their entirety.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul	Carry-less multiply (low-part)
✓	✓	clmulh	Carry-less multiply (high-part)
✓	✓	xperm4	Crossbar permutation (nibbles)
✓	✓	xperm8	Crossbar permutation (bytes)
✓	✓	ror	Rotate right (Register)
✓	✓	rol	Rotate left (Register)
✓	✓	rori	Rotate right (Immediate)
	✓	rorw	Rotate right Word (Register)
	✓	rolw	Rotate Left Word (Register)
	✓	roriw	Rotate right Word (Immediate)
✓	✓	andn	AND with inverted operand
✓	✓	orn	OR with inverted operand
✓	✓	xnor	Exclusive NOR
✓	✓	pack	Pack low halves of registers
✓	✓	packh	Pack low bytes of registers
	✓	packw	Pack low 16-bits of registers (RV64)
✓	✓	brev8	Reverse bits in bytes
✓	✓	rev8	Byte-reverse register
✓		zip	Bit interleave
✓		unzip	Bit deinterleave

32.6. Instruction Rationale

This section contains various rationale, design notes and usage recommendations for the instructions in the scalar cryptography extension. It also tries to record how the designs of instructions were derived, or where they were contributed from.

32.6.1. AES Instructions

The 32-bit instructions were derived from work in (Saarinen, 2020) and contributed to the RISC-V cryptography extension. The 64-bit instructions were developed collaboratively by task group members on our mailing list.

Supporting material, including rationale and a design space exploration for all of the AES instructions in the specification can be found in the paper "*The design of scalar AES Instruction Set Extensions for RISC-V*" (Marshall et al., 2020).

32.6.2. SHA2 Instructions

These instructions were developed based on academic work at the University of Bristol as part of the XCrypto project (Marshall et al., 2019), and contributed to the RISC-V cryptography extension.

The RV32 SHA2-512 instructions were based on this work, and developed in (Saarinen, 2020), before being contributed in the same way.

32.6.3. SM3 and SM4 Instructions

The SM4 instructions were derived from work in (Saarinen, 2020), and are hence very similar to the RV32 AES instructions.

The SM3 instructions were inspired by the SHA2 instructions, and based on development work done in (Saarinen, 2020), before being contributed to the RISC-V cryptography extension.

32.6.4. Bitmanip Instructions for Cryptography

Many of the primitive operations used in symmetric key cryptography and cryptographic hash functions are well supported by the RISC-V Bitmanip extensions (see Chapter 30).

This section repeats much of the information in Zbkb, Zbkc and Zbkx, but includes more rationale.

We proposed that the scalar cryptographic extension *reuse* a subset of the instructions from the Bitmanip extensions `Zb[abc]` directly. Specifically, this would mean that a core implementing *either* the scalar cryptographic extensions, *or* the `Zb[abc]`, *or* both, would be required to implement these instructions.

32.6.4.1. Rotations

```
RV32, RV64:                          RV64 only:
    ror    rd, rs1, rs2                  rorw   rd, rs1, rs2
    rol    rd, rs1, rs2                  rolw   rd, rs1, rs2
    rori   rd, rs1, imm                  roriw  rd, rs1, imm
```

See Bit-manipulation for Cryptography for details of these instructions.

Notes to software developers

Standard bitwise rotation is a primitive operation in many block ciphers and hash functions; it features particularly in the ARX (Add, Rotate, Xor) class of block ciphers and stream ciphers.

- *Algorithms making use of 32-bit rotations: SHA256, AES (Shift Rows), ChaCha20, SM3.*
- *Algorithms making use of 64-bit rotations: SHA512, SHA3.*

32.6.4.2. Bit & Byte Permutations

```
RV32, RV64:
    brev8   rd, rs1
    rev8    rd, rs1
```

See Bit-manipulation for Cryptography for details of these instructions.

Notes to software developers

Reversing bytes in words is very common in cryptography when setting a standard endianness for input and output data. Bit reversal within bytes is used for implementing the GHASH component of Galois/Counter Mode (GCM) (Dworkin, 2007).

```
RV32:
    zip     rd, rs1
    unzip   rd, rs1
```

See Bit-manipulation for Cryptography for details of these instructions.

Notes to software developers

These instructions perform a bit-interleave (or de-interleave) operation, and are useful for implementing the 64-bit rotations in the SHA3 (NIST, 2015) algorithm on a 32-bit architecture. On RV64, the relevant operations in SHA3 can be done natively using rotation instructions, so `zip` *and* `unzip` *are not required.*

32.6.4.3. Carry-less Multiply

```
RV32, RV64:
    clmul  rd, rs1, rs2
    clmulh rd, rs1, rs2
```

See Carry-less multiplication for Cryptography for details of these instructions. See Section 32.5 for additional implementation requirements for these instructions, related to data independent execution latency.

Notes to software developers

As is mentioned there, obvious cryptographic use-cases for carry-less multiply are for Galois Counter Mode (GCM) block cipher operations. GCM is recommended by NIST as a block cipher mode of operation (Dworkin, 2007), and is the only required mode for the TLS 1.3 protocol.

32.6.4.4. Logic With Negate

```
RV32, RV64:
    andn rd, rs1, rs2
     orn rd, rs1, rs2
    xnor rd, rs1, rs2
```

See Bit-manipulation for Cryptography for details of these instructions. These instructions are useful inside hash functions, block ciphers and for implementing software based side-channel countermeasures like masking. The `andn` instruction is also useful for constant time word-select in systems without the ternary Bitmanip `cmov` instruction.

Notes to software developers

In the context of Cryptography, these instructions are useful for: SHA3/Keccak Chi step, Bit-sliced function implementations, Software based power/EM side-channel countermeasures based on masking.

32.6.4.5. Packing

```
RV32, RV64:                          RV64:
    pack   rd, rs1, rs2                  packw  rd, rs1, rs2
    packh  rd, rs1, rs2
```

See Bit-manipulation for Cryptography for details of these instructions.

Notes to software developers

The `pack*` *instructions are useful for re-arranging halfwords within words, and generally getting data into the right shape prior to applying transforms. This is particularly useful for cryptographic algorithms which pass inputs around as (potentially unaligned) byte strings, but can operate on words made out of those byte strings. This occurs (for example) in AES when loading blocks and keys (which may not be word aligned) into registers to perform the round functions.*

32.6.4.6. Crossbar Permutation Instructions

```
RV32, RV64:
    xperm4 rd, rs1, rs2
    xperm8 rd, rs1, rs2
```

See Crossbar permutations for a complete description of these instructions.

The `xperm4` instruction operates on nibbles. `GPR[rs1]` contains a vector of `XLEN/4` 4-bit elements. `GPR[rs2]` contains a vector of `XLEN/4` 4-bit indexes. The result is each element in `GPR[rs2]` replaced by the indexed element in `GPR[rs1]`, or zero if the index into `GPR[rs2]` is out of bounds.

The `xperm8` instruction operates on bytes. `GPR[rs1]` contains a vector of `XLEN/8` 8-bit elements. `GPR[rs2]` contains a vector of `XLEN/8` 8-bit indexes. The result is each element in `GPR[rs2]` replaced by the indexed element in `GPR[rs1]`, or zero if the index into `GPR[rs2]` is out of bounds.

Notes to software developers

The instruction can be used to implement arbitrary bit permutations. For cryptography, they can accelerate bit-sliced implementations, permutation layers of block ciphers, masking based countermeasures and SBox operations.

Lightweight block ciphers using 4-bit SBoxes include: PRESENT (Bogdanov et al., 2007), Rectangle (Zhang et al., 2015), GIFT (Banik et al., 2017), Twine (Suzaki et al., 2012), Skinny, MANTIS (Beierle et al., 2016), Midori (Banik et al., 2015).

National ciphers using 8-bit SBoxes include: Camellia (Aoki et al., 2000) (Japan), Aria (Kwon et al., 2003) (Korea), AES (NIST, 2001) (USA, Belgium), SM4 (GB/T 32907-2016: SM4 Block Cipher Algorithm, 2016) (China) Kuznyechik (Russia).

All of these SBoxes can be implemented efficiently, in constant time, using the `xperm8` *instruction [1]. Note that this technique is also suitable for masking based side-channel countermeasures.*

32.7. Entropy Source Rationale and Recommendations

This **non-normative** appendix focuses on the rationale, security, self-certification, and implementation aspects of entropy sources. Hence we also discuss non-ISA system features that may be needed for cryptographic standards compliance and security testing.

32.7.1. Checklists for Design and Self-Certification

The security of cryptographic systems is based on secret bits and keys. These bits need to be random and originate from cryptographically secure Random Bit Generators (RBGs). An Entropy Source (ES) is required to construct secure RBGs.

While entropy source implementations do not have to be certified designs, RISC-V expects that they behave in a compatible manner and do not create unnecessary security risks to users. Self-evaluation and testing following appropriate security standards is usually needed to achieve this.

- **ISA Architectural Tests.** Verify, to the extent possible, that RISC-V ISA requirements in this specification are correctly implemented. This includes the state transitions (Section 32.4 and Section 32.7.6), access control (Section 32.4.3), and that `seed` ES16 `entropy` words can only be read destructively. The scope of RISC-V ISA architectural tests are those behaviors that are independent of the physical entropy source details. A smoke test ES module may be helpful in design phase.
- **Technical justification for entropy.** This may take the form of a stochastic model or a heuristic argument that explains why the noise source output is from a random, rather than pseudorandom (deterministic) process, and is not easily predictable or externally observable. A complete physical model is not necessary; research literature can be cited. For example, one can show that a good ring oscillator noise derives an amount of physical entropy from local, spontaneously occurring Johnson-Nyquist thermal noise (Saarinen, 2021), and is therefore not merely "random-looking".
- **Entropy Source Design Review.** An entropy source is more than a noise source, and must have features such as health tests (Section 32.7.4), a conditioner (Section 32.7.2.2), and a security boundary with clearly defined interfaces. One may tabulate the SHALL statements of SP 800-90B (Turan et al., 2018), FIPS 140-3 Implementation Guidance (NIST & CCCS, 2021), AIS-31 (Killmann & Schindler, 2011) or other standards being used. Official and non-official checklist tables are available: github.com/usnistgov/90B-Shall-Statements
- **Experimental Tests.** The raw noise source is subjected to entropy estimation as defined in NIST 800-90B, Section 3 (Turan et al., 2018). The interface described in Section 32.7.6 can used be to record datasets for this purpose. One also needs to show experimentally that the conditioner and health test components work appropriately to meet the ES16 output entropy requirements of Section 32.4.2. For SP 800-90B, NIST has made a min-entropy estimation package freely available: github.com/usnistgov/SP800-90B_EntropyAssessment
- **Resilience.** Above physical engineering steps should consider the operational environment of the device, which may be unexpected or hostile (actively attempting to exploit vulnerabilities in the design).

See Section 32.7.5 for a discussion of various implementation options.

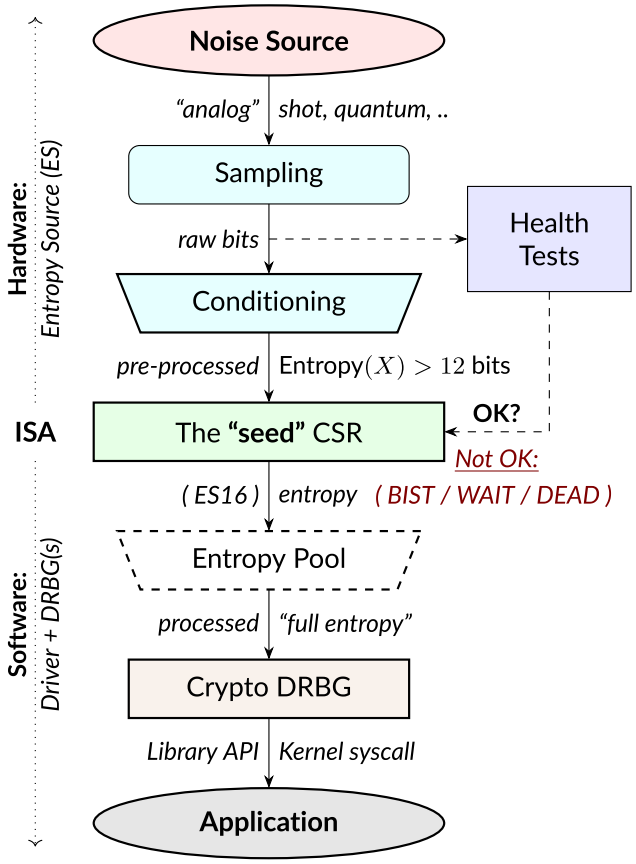
It is one of the goals of the RISC-V Entropy Source specification that a standard 90B Entropy Source Module or AIS-31 RNG IP may be licensed from a third party and integrated with a RISC-V processor design. Compared to older (FIPS 140-2) RNG and DRBG modules, an entropy source module may have a relatively small area (just a few thousand NAND2 gate equivalent). CMVP is introducing an "Entropy Source Validation Scope" which potentially allows 90B validations to be reused for different (FIPS 140-3) modules.

32.7.2. Standards and Terminology

As a fundamental security function, the generation of random numbers is governed by numerous standards and technical evaluation methods, the main ones being FIPS 140-3 (NIST, 2019; NIST & CCCS, 2021) required for U.S. Federal use, and Common Criteria Methodology (Criteria, 2017) used in high-security evaluations internationally.

Note that FIPS 140-3 is a significantly updated standard compared to its predecessor FIPS 140-2 and is only coming into use in the 2020s.

These standards set many of the technical requirements for the RISC-V entropy source design, and we use their terminology if possible.



The `seed` CSR provides an Entropy Source (ES) interface, not a stateful random number generator. As a result, it can support arbitrary security levels. Cryptographic (AES, SHA-2/3) ISA Extensions can be used to construct high-speed DRBGs that are seeded from the entropy source.

32.7.2.1. Entropy Source (ES)

Entropy sources are built by sampling and processing data from a noise source (Section 32.7.5.1). We will only consider physical sources of true randomness in this work. Since these are directly based on natural phenomena and are subject to environmental conditions (which may be adversarial), they require features that monitor the "health" and quality of those sources.

The requirements for physical entropy sources are specified in NIST SP 800-90B (Turan et al., 2018) (Section 32.4.2.1) for U.S. Federal FIPS 140-3 (NIST, 2019) evaluations and in BSI AIS-31 (Killmann & Schindler, 2001; Killmann & Schindler, 2011) (Section 32.4.2.2) for high-security Common Criteria evaluations. There is some divergence in the types of health tests and entropy metrics mandated in these standards, and RISC-V enables support for both alternatives.

32.7.2.2. Conditioning: Cryptographic and Non-Cryptographic

Raw physical randomness (noise) sources are rarely statistically perfect, and some generate very large amounts of bits, which need to be "debiased" and reduced to a smaller number of bits. This process is called conditioning. A secure hash function is an example of a cryptographic conditioner. It is important to note that even though hashing may make any data look random, it does not increase its entropy content.

Non-cryptographic conditioners and extractors such as von Neumann's "debiased coin tossing" (von Neumann, 1951) are easier to implement efficiently but may reduce entropy content (in individual bits removed) more than cryptographic hashes, which mix the input entropy very efficiently. However, they do not require cryptanalytic or computational hardness assumptions and are therefore inherently more future-proof. See Section 32.7.5.5 for a more detailed discussion.

32.7.2.3. Pseudorandom Number Generator (PRNG)

Pseudorandom Number Generators (PRNGs) use deterministic mathematical formulas to create abundant random numbers from a smaller amount of "seed" randomness. PRNGs are also divided into cryptographic and non-cryptographic ones.

Non-cryptographic PRNGs, such as LFSRs and the linear-congruential generators found in many programming libraries, may generate statistically satisfactory random numbers but must never be used for cryptographic keying. This is because they are not designed to resist *cryptanalysis*; it is usually possible to take some output and mathematically derive the "seed" or the internal state of the PRNG from it. This is a security problem since knowledge of the state allows the attacker to compute future or past outputs.

32.7.2.4. Deterministic Random Bit Generator (DRBG)

Cryptographic PRNGs are also known as Deterministic Random Bit Generators (DRBGs), a term used by SP 800-90A (Barker & Kelsey, 2015). A strong cryptographic algorithm such as AES (NIST, 2001) or SHA-2/3 (NIST, 2015; NIST, 2015) is used to produce random bits from a seed. The secret seed material is like a cryptographic key; determining the seed from the DRBG output is as hard as breaking AES or a strong hash function. This also illustrates that the seed/key needs to be long enough and come from a trusted Entropy Source. The DRBG should still be frequently refreshed (reseeded) for forward and backward security.

32.7.3. Specific Rationale and Considerations

32.7.3.1. The `seed` CSR

See Section 32.4.1.

The interface was designed to be simple so that a vendor- and device-independent driver component (e.g., in Linux kernel, embedded firmware, or a cryptographic library) may use `seed` to generate truly random bits.

An entropy source does not require a high-bandwidth interface; a single DRBG source initialization only requires 512 bits (256 bits of entropy), and DRBG output can be shared by any number of callers. Once initiated, a DRBG requires new entropy only to mitigate the risk of state compromise.

From a security perspective, it is essential that the side effect of flushing the secret entropy bits occurs upon reading. Hence we mandate a write operation on this particular CSR.

A blocking instruction may have been easier to use, but most users should be querying a (D)RBG instead of an entropy source. Without a polling-style mechanism, the entropy source could hang for thousands of

cycles under some circumstances. A `wfi` or `pause` mechanism (at least potentially) allows energy-saving sleep on MCUs and context switching on higher-end CPUs.

The reason for the particular `OPST = seed[31:0]` two-bit mechanism is to provide redundancy. The "fault" bit combinations `11` (`DEAD`) and `00` (`BIST`) are more likely for electrical reasons if feature discovery fails and the entropy source is actually not available.

The 16-bit bandwidth was a compromise motivated by the desire to provide redundancy in the return value, some protection against potential Power/EM leakage (further alleviated by the 2:1 cryptographic conditioning discussed in Section 32.7.5.6), and the desire to have all of the bits "in the same place" on both RV32 and RV64 architectures for programming convenience.

32.7.3.2. NIST SP 800-90B

See Section 32.4.2.1.

SP 800-90C (Barker et al., 2021) states that each conditioned block of n bits is required to have n+64 bits of input entropy to attain full entropy. Hence NIST SP 800-90B (Turan et al., 2018) min-entropy assessment must guarantee at least 128 + 64 = 192 bits input entropy per 256-bit block ((Barker et al., 2021), Sections 4.1. and 4.3.2). Only then a hashing of 16 * 16 = 256 bits from the entropy source will produce the desired 128 bits of full entropy. This follows from the specific requirements, threat model, and distinguishability proof contained in SP 800-90C (Barker et al., 2021), Appendix A. The implied min-entropy rate is 192/256=12/16=0.75. The expected Shannon entropy is much larger.

In FIPS 140-3 / SP 800-90 classification, an RBG2(P) construction is a cryptographically secure RBG with continuous access to a physical entropy source (`seed`) and output generated by a fully seeded, secure DRBG. The entropy source can also be used to build RBG3 full entropy sources (Barker et al., 2021). The concatenation of output words corresponds to the `Get_ES_Bitstring` function.

The 128-bit output block size was selected because that is the output size of the CBC-MAC conditioner specified in Appendix F of (Turan et al., 2018) and also the smallest key size we expect to see in applications.

If NIST SP 800-90B certification is chosen, the entropy source should implement at least the health tests defined in Section 4.4 of (Turan et al., 2018): the repetition count test and adaptive proportion test, or show that the same flaws will be detected by vendor-defined tests.

32.7.3.3. BSI AIS-31

See Section 32.4.2.2.

PTG.2 is one of the security and functionality classes defined in BSI AIS 20/31 (Killmann & Schindler, 2011). The PTG.2 source requirements work as a building block for other types of BSI generators (e.g., DRBGs, or PTG.3 TRNG with appropriate software post-processing).

For validation purposes, the PTG.2 requirements may be mapped to security controls T1-3 (Section 32.7.4) and the interface as follows:

- P1 **[PTG.2.1]** Start-up tests map to T1 and reset-triggered (on-demand) `BIST` tests.
- P2 **[PTG.2.2]** Continuous testing total failure maps to T2 and the `DEAD` state.
- P3 **[PTG.2.3]** Online tests are continuous tests of T2 – entropy output is prevented in the `BIST` state.
- P4 **[PTG.2.4]** Is related to the design of effective entropy source health tests, which we encourage.

- P5 **[PTG.2.5]** Raw random sequence may be checked via the GetNoise interface (Section 32.7.6).
- P6 **[PTG.2.6]** Test Procedure A (Killmann & Schindler, 2011) (Sect 2.4.4.1) is a part of the evaluation process, and we suggest self-evaluation using these tests even if AIS-31 certification is not sought.
- P7 **[PTG.2.7]** Average Shannon entropy of "internal random bits" exceeds 0.997.

Note how P7 concerns Shannon entropy, not min-entropy as with NIST sources. Hence the min-entropy requirement needs to be also stated. PTG.2 modules built and certified to the AIS-31 standard can also meet the "full entropy" condition after 2:1 cryptographic conditioning, but not necessarily so. The technical validation process is somewhat different.

32.7.3.4. Virtual Sources

Section 32.4.2.3.

All sources that are not direct physical sources (meeting the SP 800-90B or the AIS-31 PTG.2 requirements) need to meet the security requirements of virtual entropy sources. It is assumed that a virtual entropy source is not a limiting, shared bandwidth resource (but a software DRBG).

DRBGs can be used to feed other (virtual) DRBGs, but that does not increase the absolute amount of entropy in the system. The entropy source must be able to support current and future security standards and applications. The 256-bit requirement maps to "Category 5" of NIST Post-Quantum Cryptography (4.A.5 "Security Strength Categories" in (NIST, 2016)) and TOP SECRET schemes in Suite B and the newer U.S. Government CNSA Suite (NSA/CSS, 2015).

32.7.3.5. Security Considerations for Direct Hardware Access

Section 32.4.3.

The ISA implementation and system design must try to ensure that the hardware-software interface minimizes avenues for adversarial information flow even if not explicitly forbidden in the specification.

For security, virtualization requires both conditioning and DRBG processing of physical entropy output. It is recommended if a single physical entropy source is shared between multiple different virtual machines or if the guest OS is untrusted. A virtual entropy source is significantly more resistant to depletion attacks and also lessens the risk from covert channels.

The direct `mseccfg.[s,u]seed` option allows one to draw a security boundary around a component in relation to Sensitive Security Parameter (SSP) flows, even if that component is not in M mode. This is helpful when implementing trusted enclaves. Such modules can enforce the entire key lifecycle from birth (in the entropy source) to death (zeroization) to occur without the key being passed across the boundary to external code.

Depletion. Active polling may deny the entropy source to another simultaneously running consumer. This can (for example) delay the instantiation of that virtual machine if it requires entropy to initialize fully.

Covert Channels. Direct access to a component such as the entropy source can be used to establish communication channels across security boundaries. Active polling from one consumer makes the resource unavailable WAIT instead of ES16 to another (which is polling infrequently). Such interactions can be used to establish low-bandwidth channels.

Hardware Fingerprinting. An entropy source (and its noise source circuits) may have a uniquely identifiable hardware "signature." This can be harmless or even useful in some applications (as random sources may exhibit Physically Un-clonable Function (PUF) -like features) but highly undesirable in others

(anonymized virtualized environments and enclaves). A DRBG masks such statistical features.

Side Channels. Some of the most devastating practical attacks against real-life cryptosystems have used inconsequential-looking additional information, such as padding error messages (Bardou et al., 2012) or timing information (Moghimi et al., 2020).

We urge implementers against creating unnecessary information flows via status or custom bits or to allow any other mechanism to disable or affect the entropy source output. All information flows and interaction mechanisms must be considered from an adversarial viewpoint: the fewer the better.

As an example of side-channel analysis, we note that the entropy polling interface is typically not "constant time." One needs to analyze what kind of information is revealed via the timing oracle; one way of doing it is to model `seed` as a rejection sampler. Such a timing oracle can reveal information about the noise source type and entropy source usage, but not about the random output `entropy` bits themselves. If it does, additional countermeasures are necessary.

32.7.4. Security Controls and Health Tests

The primary purpose of a cryptographic entropy source is to produce secret keying material. In almost all cases, a hardware entropy source must implement appropriate *security controls* to guarantee unpredictability, prevent leakage, detect attacks, and deny adversarial control over the entropy output or ts generation mechanism. Explicit security controls are required for security testing and certification.

Many of the security controls built into the device are called "health checks." Health checks can take the form of integrity checks, start-up tests, and on-demand tests. These tests can be implemented in hardware or firmware, typically both. Several are mandated by standards such as NIST SP 800-90B (NIST, 2019). The choice of appropriate health tests depends on the certification target, system architecture, threat model, entropy source type, and other factors.

Health checks are not intended for hardware diagnostics but for detecting security issues. Hence the default action in case of a failure should be aimed at damage control: Limiting further output and preventing weak crypto keys from being generated.

We discuss three specific testing requirements T1-T3. The testing requirement follows from the definition of an Entropy Source; without it, the module is simply a noise source and can't be trusted to safely generate keying material.

32.7.4.1. T1: On-demand testing

A sequence of simple tests is invoked via resetting, rebooting, or powering up the hardware (not an ISA signal). The implementation will simply return `BIST` during the initial start-up self-test period; in any case, the driver must wait for them to finish before starting cryptographic operations. Upon failure, the entropy source will enter a no-output `DEAD` state.

Rationale. Interaction with hardware self-test mechanisms from the software side should be minimal; the term "on-demand" does not mean that the end-user or application program should be able to invoke them in the field (the term is a throwback to an age of discrete, non-autonomous crypto devices with human operators).

32.7.4.2. T2: Continuous checks

If an error is detected in continuous tests or environmental sensors, the entropy source will enter a no-output state. We define that a non-critical alarm is signaled if the entropy source returns to `BIST` state from

live (`WAIT` or `ES16`) states. Critical failures will result in `DEAD` state immediately. A hardware-based continuous testing mechanism must not make statistical information externally available, and it must be zeroized periodically or upon demand via reset, power-up, or similar signal.

Rationale. Physical attacks can occur while the device is running. The design should avoid guiding such active attacks by revealing detailed status information. Upon detection of an attack, the default action should be aimed at damage control — to prevent weak crypto keys from being generated.

The statistical nature of some tests makes "type-1" false positives a possibility. There may also be requirements for signaling of non-fatal alarms; AIS 31 specifies "noise alarms" that can go off with non-negligible probability even if the device is functioning correctly; these can be signaled with `BIST`. There rarely is anything that can or should be done about a non-fatal alarm condition in an operator-free, autonomous system.

The state of statistical runtime health checks (such as counters) is potentially correlated with some secret keying material, hence the zeroization requirement.

32.7.4.3. T3: Fatal error states

Since the security of most cryptographic operations depends on the entropy source, a system-wide "default deny" security policy approach is appropriate for most entropy source failures. A hardware test failure should at least result in the `DEAD` state and possibly reset/halt. It's a show stopper: The entropy source (or its cryptographic client application) *must not* be allowed to run if its secure operation can't be guaranteed.

Rationale. These tests can complement other integrity and tamper resistance mechanisms (See Chapter 18 of (Anderson, 2020) for examples).

Some hardware random generators are, by their physical construction, exposed to relatively non-adversarial environmental and manufacturing issues. However, even such "innocent" failure modes may indicate a *fault attack* (Karaklajic et al., 2013) and therefore should be addressed as a system integrity failure rather than as a diagnostic issue.

Security architects will understand to use permanent or hard-to-recover "security-fuse" lockdowns only if the threshold of a test is such that the probability of false-positive is negligible over the entire device lifetime.

32.7.4.4. Information Flows

Some of the most devastating practical attacks against real-life cryptosystems have used inconsequential-looking additional information, such as padding error messages (Bardou et al., 2012) or timing information (Moghimi et al., 2020). In cryptography, such out-of-band information sources are called "oracles."

To guarantee that no sensitive data is read twice and that different callers don't get correlated output, it is required that hardware implements *wipe-on-read* on the randomness pathway during each read (successful poll). For the same reasons, only complete and fully processed random words shall be made available via `entropy` (ES16 status of `seed`).

This also applies to the raw noise source. The raw source interface has been delegated to an optional vendor-specific test interface. Importantly the test interface and the main interface should not be operational at the same time.

> The noise source state shall be protected from adversarial knowledge or influence to the greatest extent possible. The methods used for this shall be documented, including a description of the (conceptual) security boundary's role in protecting the

> noise source from adversarial observation or influence.
>
> — NIST SP 800-90B, Noise Source Requirements

An entropy source is a singular resource, subject to depletion and also covert channels (Evtyushkin & Ponomarev, 2016). Observation of the entropy can be the same as the observation of the noise source output, as cryptographic conditioning is mandatory only as a post-processing step. SP 800-90B and other security standards mandate protection of noise bits from observation and also influence.

32.7.5. Implementation Strategies

As a general rule, RISC-V specifies the ISA only. We provide some additional suggestions so that portable, vendor-independent middleware and kernel components can be created. The actual hardware implementation and certification are left to vendors and circuit designers; the discussion in this Section is purely informational.

When considering implementation options and trade-offs, one must look at the entire information flow.

1. **A Noise Source** generates private, unpredictable signals from stable and well-understood physical random events.
2. **Sampling** digitizes the noise signal into a raw stream of bits. This raw data also needs to be protected by the design.
3. **Continuous health tests** ensure that the noise source and its environment meet their operational parameters.
4. **Non-cryptographic conditioners** remove much of the bias and correlation in input noise.
5. **Cryptographic conditioners** produce full entropy output, completely indistinguishable from ideal random.
6. **DRBG** takes in `>=256` bits of seed entropy as keying material and uses a "one way" cryptographic process to rapidly generate bits on demand (without revealing the seed/state).

Steps 1-4 (possibly 5) are considered to be part of the Entropy Source (ES) and provided by the `seed` CSR. Adding the software-side cryptographic steps 5-6 and control logic complements it into a True Random Number Generator (TRNG).

32.7.5.1. Ring Oscillators

We will give some examples of common noise sources that can be implemented in the processor itself (using standard cells).

The most common entropy source type in production use today is based on "free running" ring oscillators and their timing jitter. Here, an odd number of inverters is connected into a loop from which noise source bits are sampled in relation to a reference clock (Baudet et al., 2011). The sampled bit sequence may be expected to be relatively uncorrelated (close to IID) if the sample rate is suitably low (Killmann & Schindler, 2011). However, further processing is usually required.

AMD (AMD, 2017), ARM (ARM, 2017), and IBM (Liberty et al., 2013) are examples of ring oscillator TRNGs intended for high-security applications.

There are related metastability-based generator designs such as Transition Effect Ring Oscillator (TERO) (Varchola & Drutarovský, 2010). The differential/feedback Intel construction (Hamburg et al., 2012) is slightly different but also falls into the same general metastable oscillator-based category.

The main benefits of ring oscillators are: (1) They can be implemented with standard cell libraries without external components — and even on FPGAs (Valtchanov et al., 2010), (2) there is an established theory for their behavior (Hajimiri & Lee, 1998; Hajimiri et al., 1999; Baudet et al., 2011), and (3) ample precedent exists for testing and certifying them at the highest security levels.

Ring oscillators also have well-known implementation pitfalls. Their output is sometimes highly dependent on temperature, which must be taken into account in testing and modeling. If the ring oscillator construction is parallelized, it is important that the number of stages and/or inverters in each chain is suitable to avoid entropy reduction due to harmonic "Huyghens synchronization" (Bak, 1986). Such harmonics can also be inserted maliciously in a frequency injection attack, which can have devastating results (Markettos & Moore, 2009). Countermeasures are related to circuit design; environmental sensors, electrical filters, and usage of a differential oscillator may help.

32.7.5.2. Shot Noise

A category of random sources consisting of discrete events and modeled as a Poisson process is called "shot noise." There's a long-established precedent of certifying them; the AIS 31 document (Killmann & Schindler, 2011) itself offers reference designs based on noisy diodes. Shot noise sources are often more resistant to temperature changes than ring oscillators. Some of these generators can also be fully implemented with standard cells (The Rambus / Inside Secure generic TRNG IP (Rambus, 2020) is described as a Shot Noise generator).

32.7.5.3. Other types of noise

It may be possible to certify more exotic noise sources and designs, although their stochastic model needs to be equally well understood, and their CPU interfaces must be secure. See Section 32.7.5.8 for a discussion of Quantum entropy sources.

32.7.5.4. Continuous Health Tests

Health monitoring requires some state information related to the noise source to be maintained. The tests should be designed in a way that a specific number of samples guarantees a state flush (no hung states). We suggest flush size `W =< 1024` to match with the NIST SP 800-90B required tests (See Section 4.4 in (Turan et al., 2018)). The state is also fully zeroized in a system reset.

The two mandatory tests can be built with minimal circuitry. Full histograms are not required, only simple counter registers: repetition count, window count, and sample count. Repetition count is reset every time the output sample value changes; if the count reaches a certain cutoff limit, a noise alarm (`BIST`) or failure (`DEAD`) is signaled. The window counter is used to save every W'th output (typically `W` in { 512, 1024 }). The frequency of this reference sample in the following window is counted; cutoff values are defined in the standard. We see that the structure of the mandatory tests is such that, if well implemented, no information is carried beyond a limit of `W` samples.

Section 4.5 of (Turan et al., 2018) explicitly permits additional developer-defined tests, and several more were defined in early versions of FIPS 140-1 before being "crossed out." The choice of additional tests depends on the nature and implementation of the physical source.

Especially if a non-cryptographic conditioner is used in hardware, it is possible that the AIS 31 (Killmann & Schindler, 2011) online tests are implemented by driver software. They can also be implemented in hardware. For some security profiles, AIS 31 mandates that their tolerances are set in a way that the probability of an alarm is at least 10^{-6} yearly under "normal usage." Such requirements are problematic in modern applications since their probability is too high for critical systems.

There rarely is anything that can or should be done about a non-fatal alarm condition in an operator-free, autonomous system. However, AIS 31 allows the DRBG component to keep running despite a failure in its Entropy Source, so we suggest re-entering a temporary `BIST` state (Section 32.7.4) to signal a non-fatal statistical error if such (non-actionable) signaling is necessary. Drivers and applications can react to this appropriately (or simply log it), but it will not directly affect the availability of the TRNG. A permanent error condition should result in `DEAD` state.

32.7.5.5. Non-cryptographic Conditioners

As noted in Section 32.7.2.2, physical randomness sources generally require a post-processing step called *conditioning* to meet the desired quality requirements, which are outlined in Section 32.4.2.

The approach taken in this interface is to allow a combination of non-cryptographic and cryptographic filtering to take place. The first stage (hardware) merely needs to be able to distill the entropy comfortably above the necessary level.

- One may take a set of bits from a noise source and XOR them together to produce a less biased (and more independent) bit. However, such an XOR may introduce "pseudorandomness" and make the output difficult to analyze.
- The von Neumann extractor (von Neumann, 1951) looks at consecutive pairs of bits, rejects 00 and 11, and outputs 0 or 1 for 01 and 10, respectively. It will reduce the number of bits to less than 25% of the original, but the output is provably unbiased (assuming independence).
- Blum's extractor (Blum, 1986) can be used on sources whose behavior resembles N-state Markov chains. If its assumptions hold, it also removes dependencies, creating an independent and identically distributed (IID) source.
- Other linear and non-linear correctors such as those discussed by Dichtl and Lacharme (Lacharme, 2008).

Note that the hardware may also implement a full cryptographic conditioner in the entropy source, even though the software driver still needs a cryptographic conditioner, too (Section 32.4.2).

Rationale: The main advantage of non-cryptographic extractors is in their energy efficiency, relative simplicity, and amenability to mathematical analysis. If well designed, they can be evaluated in conjunction with a stochastic model of the noise source itself. They do not require computational hardness assumptions.

32.7.5.6. Cryptographic Conditioners

For secure use, cryptographic conditioners are always required on the software side of the ISA boundary. They may also be implemented on the hardware side if necessary. In any case, the `entropy` ES16 output must always be compressed 2:1 (or more) before being used as keying material or considered "full entropy."

Examples of cryptographic conditioners include the random pool of the Linux operating system, secure hash functions (SHA-2/3, SHAKE (NIST, 2015; NIST, 2015)), and the AES / CBC-MAC construction in Appendix F, SP 800-90B (Turan et al., 2018).

In some constructions, such as the Linux RNG and SHA-3/SHAKE (NIST, 2015) based generators, the cryptographic conditioning and output (DRBG) generation are provided by the same component.

Rationale: For many low-power targets constructions the type of hardware AES CBC-MAC conditioner used by Intel (Mechalas, 2018) and AMD (AMD, 2017) would be too complex and energy-hungry to implement solely to serve the `seed` CSR. On the other hand, simpler non-cryptographic conditioners may be too wasteful on input entropy if high-quality random output is required — (ARM TrustZone TRBG

(ARM, 2017) outputs only 10Kbit/sec at 200 MHz.) Hence a resource-saving compromise is made between hardware and software generation.

32.7.5.7. The Final Random: DRBGs

All random bits reaching end users and applications must come from a cryptographic DRBG. These are generally implemented by the driver component in software. The RISC-V AES and SHA instruction set extensions should be used if available since they offer additional security features such as timing attack resistance.

Currently recommended DRBGs are defined in NIST SP 800-90A (Rev 1) (Barker & Kelsey, 2015): `CTR_DRBG`, `Hash_DRBG`, and `HMAC_DRBG`. Certification often requires known answer tests (KATs) for the symmetric components and the DRBG as a whole. These are significantly easier to implement in software than in hardware. In addition to the directly certifiable SP 800-90A DRBGs, a Linux-style random pool construction based on ChaCha20 (Müller, 2020) can be used, or an appropriate construction based on SHAKE256 (NIST, 2015).

These are just recommendations; programmers can adjust the usage of the CPU Entropy Source to meet future requirements.

32.7.5.8. Quantum vs. Classical Random

> The NCSC believes that classical RNGs will continue to meet our needs for government and military applications for the foreseeable future.
>
> — U.K. NCSC QRNG Guidance, March 2020

A Quantum Random Number Generator (QRNG) is a TRNG whose source of randomness can be unambiguously identified to be a specific quantum phenomenon such as quantum state superposition, quantum state entanglement, Heisenberg uncertainty, quantum tunneling, spontaneous emission, or radioactive decay (ITU, 2019).

Direct quantum entropy is theoretically the best possible kind of entropy. A typical TRNG based on electronic noise is also largely based on quantum phenomena and is equally unpredictable - the difference is that the relative amount of quantum and classical physics involved is difficult to quantify for a classical TRNG.

QRNGs are designed in a way that allows the amount of quantum-origin entropy to be modeled and estimated. This distinction is important in the security model used by QKD (Quantum Key Distribution) security mechanisms which can be used to protect the physical layer (such as fiber optic cables) against interception by using quantum mechanical effects directly.

This security model means that many of the available QRNG devices do not use cryptographic conditioning and may fail cryptographic statistical requirements (Hurley-Smith & Hernández-Castro, 2020). Many implementers may consider them to be entropy sources instead.

Relatively little research has gone into QRNG implementation security, but many QRNG designs are arguably more susceptible to leakage than classical generators (such as ring oscillators) as they tend to employ external components and mixed materials. As an example, amplification of a photon detector signal may be observable in power analysis, which classical noise-based sources are designed to resist.

32.7.5.9. Post-Quantum Cryptography

PQC public-key cryptography standards (NIST, 2016) do not require quantum-origin randomness, just sufficiently secure keying material. Recall that cryptography aims to protect the confidentiality and integrity of data itself and does not place any requirements on the physical communication channel (like QKD).

Classical good-quality TRNGs are perfectly suitable for generating the secret keys for PQC protocols that are hard for quantum computers to break but implementable on classical computers. What matters in cryptography is that the secret keys have enough true randomness (entropy) and that they are generated and stored securely.

Of course, one must avoid DRBGs that are based on problems that are easily solvable with quantum computers, such as factoring (Shor, 1994) in the case of the Blum-Blum-Shub generator (Blum et al., 1986). Most symmetric algorithms are not affected as the best quantum attacks are still exponential to key size (Grover, 1996).

As an example, the original Intel RNG (Mechalas, 2018), whose output generation is based on AES-128, can be attacked using Grover's algorithm with approximately square-root effort (Jaques et al., 2020). While even "64-bit" quantum security is extremely difficult to break, many applications specify a higher security requirement. NIST (NIST, 2016) defines AES-128 to be "Category 1" equivalent post-quantum security, while AES-256 is "Category 5" (highest). We avoid this possible future issue by exposing direct access to the entropy source which can derive its security from information-theoretic assumptions only.

32.7.6. Suggested GetNoise Test Interface

Compliance testing, characterization, and configuration of entropy sources require access to raw, unconditioned noise samples. This conceptual test interface is named GetNoise in Section 2.3.2 of NIST SP 800-90B (Turan et al., 2018).

Since this type of interface is both necessary for security testing and also constitutes a potential backdoor to the cryptographic key generation process, we define a safety behavior that compliant implementations can have for temporarily disabling the entropy source `seed` CSR interface during test.

In order for shared RISC-V self-certification scripts (and drivers) to accommodate the test interface in a secure fashion, we suggest that it is implemented as a custom, M-mode only CSR, denoted here as `mnoise`.

This non-normative interface is not intended to be used as a source of randomness or for other production use. We define the semantics for single bit for this interface, `mnoise[31]`, which is named `NOISE_TEST`, which will affect the behavior of `seed` if implemented.

When `NOISE_TEST = 1` in `mnoise`, the `seed` CSR must not return anything via `ES16`; it should be in `BIST` state unless the source is `DEAD`. When `NOISE_TEST` is again disabled, the entropy source shall return from `BIST` via an appropriate zeroization and self-test mechanism.

The behavior of other input and output bits is largely left to the vendor (as they depend on the technical details of the physical entropy source), as is the address of the custom `mnoise` CSR. Other contents and behavior of the CSR only can be interpreted in the context of `mvendorid`, `marchid`, and `mimpid` CSR identifiers.

When not implemented (e.g., in virtual machines), `mnoise` can permanently read zero (`0x00000000`) and ignore writes. When available, but `NOISE_TEST = 0`, `mnoise` can return a nonzero constant (e.g. `0x00000001`) but no noise samples.

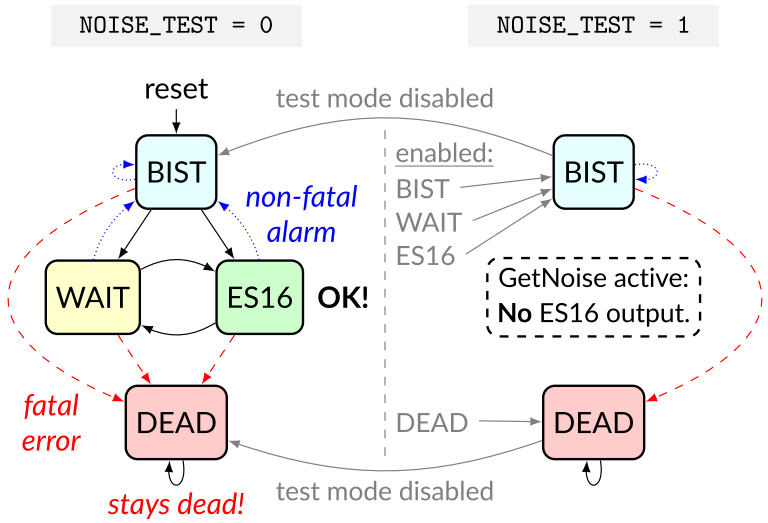


Figure 7. Entropy source can't be read in test mode.

In `NOISE_TEST` mode, the WAIT and ES16 states are unreachable, and no entropy is output. Implementation of test interfaces that directly affect ES16 entropy output from the `seed` CSR interface is discouraged. Such vendor test interfaces have been exploited in attacks. For example, an ECDSA (NIST, 2013) signature process without sufficient entropy will not only create an insecure signature but can also reveal the secret signing key, that can be used for authentication forgeries by attackers. Hence even a temporary lapse in `entropy` security may have serious security implications.

32.8. Supplementary Materials

While this document contains the specifications for the RISC-V cryptography extensions, numerous supplementary materials and example codes have also been developed. All of the materials related to the RISC-V Cryptography extension live in a GitHub Repository, located at github.com/riscv/riscv-crypto

- `doc/` Contains the source code for this document.
- `doc/supp/` Contains supplementary information and recommendations for implementers of software and hardware.
- `benchmarks/` Example software implementations.
- `rtl/` Example Verilog implementations of each instruction.
- `sail/` Formal model implementations in Sail.

32.9. Supporting Sail Code

This section contains the supporting Sail code referenced by the instruction descriptions throughout the specification. The Sail Manual is recommended reading in order to best understand the supporting code.

```
/* Auxiliary function for performing GF multiplication */
val xt2 : bits(8) -> bits(8)
function xt2(x) = {
  (x << 1) ^ (if bit_to_bool(x[7]) then 0x1b else 0x00)
}

val xt3 : bits(8) -> bits(8)
```

```
function xt3(x) = x ^ xt2(x)

/* Multiply 8-bit field element by 4-bit value for AES MixCols step */
val gfmul : (bits(8), bits(4)) -> bits(8)
function gfmul( x, y) = {
  (if bit_to_bool(y[0]) then             x    else 0x00) ^
  (if bit_to_bool(y[1]) then xt2(        x)   else 0x00) ^
  (if bit_to_bool(y[2]) then xt2(xt2(    x))  else 0x00) ^
  (if bit_to_bool(y[3]) then xt2(xt2(xt2(x))) else 0x00)
}

/* 8-bit to 32-bit partial AES Mix Column - forwards */
val aes_mixcolumn_byte_fwd : bits(8) -> bits(32)
function aes_mixcolumn_byte_fwd(so) = {
  gfmul(so, 0x3) @ so @ so @ gfmul(so, 0x2)
}

/* 8-bit to 32-bit partial AES Mix Column - inverse*/
val aes_mixcolumn_byte_inv : bits(8) -> bits(32)
function aes_mixcolumn_byte_inv(so) = {
  gfmul(so, 0xb) @ gfmul(so, 0xd) @ gfmul(so, 0x9) @ gfmul(so, 0xe)
}

/* 32-bit to 32-bit AES forward MixColumn */
val aes_mixcolumn_fwd : bits(32) -> bits(32)
function aes_mixcolumn_fwd(x) = {
  let s0 : bits (8) = x[ 7.. 0];
  let s1 : bits (8) = x[15.. 8];
  let s2 : bits (8) = x[23..16];
  let s3 : bits (8) = x[31..24];
  let b0 : bits (8) = xt2(s0) ^ xt3(s1) ^    (s2) ^    (s3);
  let b1 : bits (8) =    (s0) ^ xt2(s1) ^ xt3(s2) ^    (s3);
  let b2 : bits (8) =    (s0) ^    (s1) ^ xt2(s2) ^ xt3(s3);
  let b3 : bits (8) = xt3(s0) ^    (s1) ^    (s2) ^ xt2(s3);
  b3 @ b2 @ b1 @ b0 /* Return value */
}

/* 32-bit to 32-bit AES inverse MixColumn */
val aes_mixcolumn_inv : bits(32) -> bits(32)
function aes_mixcolumn_inv(x) = {
  let s0 : bits (8) = x[ 7.. 0];
  let s1 : bits (8) = x[15.. 8];
  let s2 : bits (8) = x[23..16];
  let s3 : bits (8) = x[31..24];
  let b0 : bits (8) = gfmul(s0, 0xE) ^ gfmul(s1, 0xB) ^ gfmul(s2, 0xD) ^ gfmul
(s3, 0x9);
  let b1 : bits (8) = gfmul(s0, 0x9) ^ gfmul(s1, 0xE) ^ gfmul(s2, 0xB) ^ gfmul
(s3, 0xD);
  let b2 : bits (8) = gfmul(s0, 0xD) ^ gfmul(s1, 0x9) ^ gfmul(s2, 0xE) ^ gfmul
(s3, 0xB);
```

```
  let b3 : bits (8) = gfmul(s0, 0xB) ^ gfmul(s1, 0xD) ^ gfmul(s2, 0x9) ^ gfmul
(s3, 0xE);
  b3 @ b2 @ b1 @ b0 /* Return value */
}

/* Turn a round number into a round constant for AES. Note that the
   AES64KS1I instruction is defined such that the r argument is always
   in the range 0x0..0xA. Values of rnum outside the range 0x0..0xA
   do not decode to the AES64KS1I instruction. The 0xA case is used
   specifically for the AES-256 KeySchedule, and this function is never
   called in that case. */
val aes_decode_rcon : bits(4) -> bits(32)
function aes_decode_rcon(r) = {
  assert(r <_u 0xA);
  match r {
    0x0 => 0x00000001,
    0x1 => 0x00000002,
    0x2 => 0x00000004,
    0x3 => 0x00000008,
    0x4 => 0x00000010,
    0x5 => 0x00000020,
    0x6 => 0x00000040,
    0x7 => 0x00000080,
    0x8 => 0x0000001b,
    0x9 => 0x00000036,
    _   => internal_error(__FILE__, __LINE__, "Unexpected AES r") /*
unreachable -- required to silence Sail warning */
  }
}

/* SM4 SBox - only one sbox for forwards and inverse */
let sm4_sbox_table : vector(256, bits(8)) = [
0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28,
0xFB, 0x2C, 0x05, 0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44,
0x13, 0x26, 0x49, 0x86, 0x06, 0x99, 0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98,
0x7A, 0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62, 0xE4, 0xB3, 0x1C, 0xA9,
0xC9, 0x08, 0xE8, 0x95, 0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6, 0x47,
0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA, 0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85,
0x4F, 0xA8, 0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B, 0xF8, 0xEB, 0x0F,
0x4B, 0x70, 0x56, 0x9D, 0x35, 0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2,
0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87, 0xD4, 0x00, 0x46, 0x57, 0x9F,
0xD3, 0x27, 0x52, 0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E, 0xEA, 0xBF,
0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5, 0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15,
0xA1, 0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55, 0xAD, 0x93, 0x32, 0x30,
0xF5, 0x8C, 0xB1, 0xE3, 0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60, 0xC0,
0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F, 0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD,
0x8E, 0x2F, 0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51, 0x8D, 0x1B, 0xAF,
0x92, 0xBB, 0xDD, 0xBC, 0x7F, 0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD, 0x2D, 0x74, 0xD0, 0x12, 0xB8,
0xE5, 0xB4, 0xB0, 0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E, 0x65, 0xB9,
```

```
0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84, 0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D,
0x20, 0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48
]

let aes_sbox_fwd_table : vector(256, bits(8)) = [
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4,
0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7,
0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3,
0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09,
0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe,
0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92,
0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c,
0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2,
0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25,
0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e,
0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42,
0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
]

let aes_sbox_inv_table : vector(256, bits(8)) = [
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e,
0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23,
0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, 0x08, 0x2e, 0xa1, 0x66,
0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, 0x72,
0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65,
0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46,
0x57, 0xa7, 0x8d, 0x9d, 0x84, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca,
0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91,
0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,
0x73, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f,
0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2,
0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, 0x1f, 0xdd, 0xa8,
0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb,
0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6,
0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
]
```

```
/* Lookup function - takes an index and a table, and retrieves the
 * x'th element of that table. Note that the Sail vector literals
 * start at index 255, and go down to 0.
 */
val sbox_lookup : (bits(8), vector(256, bits(8))) -> bits(8)
function sbox_lookup(x, table) = {
  table[255 - unsigned(x)]
}

/* Easy function to perform a forward AES SBox operation on 1 byte. */
val aes_sbox_fwd : bits(8) -> bits(8)
function aes_sbox_fwd(x) = sbox_lookup(x, aes_sbox_fwd_table)

/* Easy function to perform an inverse AES SBox operation on 1 byte. */
val aes_sbox_inv : bits(8) -> bits(8)
function aes_sbox_inv(x) = sbox_lookup(x, aes_sbox_inv_table)

/* AES SubWord function used in the key expansion
 * - Applies the forward sbox to each byte in the input word.
 */
val aes_subword_fwd : bits(32) -> bits(32)
function aes_subword_fwd(x) = {
  aes_sbox_fwd(x[31..24]) @
  aes_sbox_fwd(x[23..16]) @
  aes_sbox_fwd(x[15.. 8]) @
  aes_sbox_fwd(x[ 7.. 0])
}

/* AES Inverse SubWord function.
 * - Applies the inverse sbox to each byte in the input word.
 */
val aes_subword_inv : bits(32) -> bits(32)
function aes_subword_inv(x) = {
  aes_sbox_inv(x[31..24]) @
  aes_sbox_inv(x[23..16]) @
  aes_sbox_inv(x[15.. 8]) @
  aes_sbox_inv(x[ 7.. 0])
}

/* Easy function to perform an SM4 SBox operation on 1 byte. */
val sm4_sbox : bits(8) -> bits(8)
function sm4_sbox(x) = sbox_lookup(x, sm4_sbox_table)

val aes_get_column : (bits(128), nat) -> bits(32)
function aes_get_column(state,c) = (state >> (to_bits(7, 32 * c)))[31..0]

/* 64-bit to 64-bit function which applies the AES forward sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_fwd_sbox_to_each_byte : bits(64) -> bits(64)
```

```
function aes_apply_fwd_sbox_to_each_byte(x) = {
  aes_sbox_fwd(x[63..56]) @
  aes_sbox_fwd(x[55..48]) @
  aes_sbox_fwd(x[47..40]) @
  aes_sbox_fwd(x[39..32]) @
  aes_sbox_fwd(x[31..24]) @
  aes_sbox_fwd(x[23..16]) @
  aes_sbox_fwd(x[15.. 8]) @
  aes_sbox_fwd(x[ 7.. 0])
}

/* 64-bit to 64-bit function which applies the AES inverse sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_inv_sbox_to_each_byte : bits(64) -> bits(64)
function aes_apply_inv_sbox_to_each_byte(x) = {
  aes_sbox_inv(x[63..56]) @
  aes_sbox_inv(x[55..48]) @
  aes_sbox_inv(x[47..40]) @
  aes_sbox_inv(x[39..32]) @
  aes_sbox_inv(x[31..24]) @
  aes_sbox_inv(x[23..16]) @
  aes_sbox_inv(x[15.. 8]) @
  aes_sbox_inv(x[ 7.. 0])
}

/*
 * AES full-round transformation functions.
 */

val getbyte : (bits(64), int) -> bits(8)
function getbyte(x, i) = (x >> to_bits(6, i * 8))[7..0]

val aes_rv64_shiftrows_fwd : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_fwd(rs2, rs1) = {
  getbyte(rs1, 3) @
  getbyte(rs2, 6) @
  getbyte(rs2, 1) @
  getbyte(rs1, 4) @
  getbyte(rs2, 7) @
  getbyte(rs2, 2) @
  getbyte(rs1, 5) @
  getbyte(rs1, 0)
}

val aes_rv64_shiftrows_inv : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_inv(rs2, rs1) = {
  getbyte(rs2, 3) @
  getbyte(rs2, 6) @
  getbyte(rs1, 1) @
```

```
  getbyte(rs1, 4) @
  getbyte(rs1, 7) @
  getbyte(rs2, 2) @
  getbyte(rs2, 5) @
  getbyte(rs1, 0)
}

/* 128-bit to 128-bit implementation of the forward AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits  7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_fwd : bits(128) -> bits(128)
function aes_shift_rows_fwd(x) = {
  let ic3 : bits(32) = aes_get_column(x, 3);
  let ic2 : bits(32) = aes_get_column(x, 2);
  let ic1 : bits(32) = aes_get_column(x, 1);
  let ic0 : bits(32) = aes_get_column(x, 0);
  let oc0 : bits(32) = ic0[31..24] @ ic1[23..16] @ ic2[15.. 8] @ ic3[ 7.. 0];
  let oc1 : bits(32) = ic1[31..24] @ ic2[23..16] @ ic3[15.. 8] @ ic0[ 7.. 0];
  let oc2 : bits(32) = ic2[31..24] @ ic3[23..16] @ ic0[15.. 8] @ ic1[ 7.. 0];
  let oc3 : bits(32) = ic3[31..24] @ ic0[23..16] @ ic1[15.. 8] @ ic2[ 7.. 0];
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* 128-bit to 128-bit implementation of the inverse AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits  7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_inv : bits(128) -> bits(128)
function aes_shift_rows_inv(x) = {
  let ic3 : bits(32) = aes_get_column(x, 3); /* In column 3 */
  let ic2 : bits(32) = aes_get_column(x, 2);
  let ic1 : bits(32) = aes_get_column(x, 1);
  let ic0 : bits(32) = aes_get_column(x, 0);
  let oc0 : bits(32) = ic0[31..24] @ ic3[23..16] @ ic2[15.. 8] @ ic1[ 7.. 0];
  let oc1 : bits(32) = ic1[31..24] @ ic0[23..16] @ ic3[15.. 8] @ ic2[ 7.. 0];
  let oc2 : bits(32) = ic2[31..24] @ ic1[23..16] @ ic0[15.. 8] @ ic3[ 7.. 0];
  let oc3 : bits(32) = ic3[31..24] @ ic2[23..16] @ ic1[15.. 8] @ ic0[ 7.. 0];
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the forward sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_fwd : bits(128) -> bits(128)
function aes_subbytes_fwd(x) = {
  let oc0 : bits(32) = aes_subword_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_fwd(aes_get_column(x, 3));
```

```
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_inv : bits(128) -> bits(128)
function aes_subbytes_inv(x) = {
  let oc0 : bits(32) = aes_subword_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the forward MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_fwd : bits(128) -> bits(128)
function aes_mixcolumns_fwd(x) = {
  let oc0 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_inv : bits(128) -> bits(128)
function aes_mixcolumns_inv(x) = {
  let oc0 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}
```

[1] svn.clairexen.net/handicraft/2020/lut4perm/demo02.cc

Chapter 33. Cryptography Extensions: Vector Instructions, Version 1.0

This document describes the Vector Cryptography extensions to the RISC-V Instruction Set Architecture.

33.1. Introduction

This document describes the RISC-V *vector* cryptography extensions. All instructions described here are based on the Vector registers. The instructions are designed to be highly performant, with large application and server-class cores being the main target. A companion chapter *Volume I: Scalar & Entropy Source Instructions*, describes cryptographic instructions for smaller cores which do not implement the vector extension.

33.1.1. Intended Audience

Cryptography is a specialized subject, requiring people with many different backgrounds to cooperate in its secure and efficient implementation. Where possible, we have written this specification to be understandable by all, though we recognize that the motivations and references to algorithms or other specifications and standards may be unfamiliar to those who are not domain experts.

This specification anticipates being read and acted on by various people with different backgrounds. We have tried to capture these backgrounds here, with a brief explanation of what we expect them to know, and how it relates to the specification. We hope this aids people's understanding of which aspects of the specification are particularly relevant to them, and which they may (safely!) ignore or pass to a colleague.

Cryptographers and cryptographic software developers

These are the people we expect to write code using the instructions in this specification. They should understand the motivations for the instructions we include, and be familiar with most of the algorithms and outside standards to which we refer.

Computer architects

We do not expect architects to have a cryptography background. We nonetheless expect architects to be able to examine our instructions for implementation issues, understand how the instructions will be used in context, and advise on how best to fit the functionality the cryptographers want.

Digital design engineers & micro-architects

These are the people who will implement the specification inside a core. Again, no cryptography expertise is assumed, but we expect them to interpret the specification and anticipate any hardware implementation issues, e.g., where high-frequency design considerations apply, or where latency/area tradeoffs exist etc. In particular, they should be aware of the literature around efficiently implementing AES and SM4 SBoxes in hardware.

Verification engineers

These people are responsible for ensuring the correct implementation of the extensions in hardware. No cryptography background is assumed. We expect them to identify interesting test cases from the specification. An understanding of their real-world usage will help with this.

These are by no means the only people concerned with the specification, but they are the ones we considered most while writing it.

33.1.2. Sail Specifications

RISC-V maintains a formal model of the ISA specification, implemented in the Sail ISA specification language (*SAIL ISA Specification Language*, n.d.). Note that *Sail* refers to the specification language itself, and that there is a *model of RISC-V*, written using Sail.

It was our intention to include actual Sail code in this specification. However, the Vector Crypto Sail model needs the Vector Sail model as a basis on which to build. This Vector Cryptography extensions specification was completed before there was an approved RISC-V Vector Sail Model. Therefore, we don't have any Sail code to include in the instruction descriptions. Instead we have included Sail-like pseudocode. While we have endeavored to adhere to Sail syntax, we have taken some liberties for the sake of simplicity where we believe that that our intent is clear to the reader.

Where variables are concatenated, the order shown is how they would appear in a vector register from left to right. For example, an element group specified as `{a, b, e, f}` *would appear in a vector register with* `a` *having the highest element index of the group and* `f` *having the lowest index of the group.*

For the sake of brevity, our pseudocode does not include the handling of masks or tail elements. We follow the *undisturbed* and *agnostic* policies for masks and tails as described in the **RISC-V "V" Vector Extension** specification. Furthermore, the code does not explicitly handle overlap and SEW constraints; these are, however, explicitly stated in the text.

In many cases the pseudocode includes calls to supporting functions which are too verbose to include directly in the specification. This supporting code is listed in Section 33.6.

The Sail Manual is recommended reading in order to best understand the code snippets. Also, The Sail Programming Language: A Sail Cookbook is a good reference.

For the latest RISC-V Sail model, refer to the formal model GitHub repository.

33.1.3. Policies

In creating this extension, we tried to adhere to the following policies:

- Where there is a choice between: 1) supporting diverse implementation strategies for an algorithm or 2) supporting a single implementation style which is more performant / less expensive; the vector crypto extensions will pick the more constrained but performant option. This fits a common pattern in other parts of the RISC-V specifications, where recommended (but not required) instruction sequences for performing particular tasks are given as an example, such that both hardware and software implementers can optimize for only a single use-case.
- The extensions will be designed to support *existing* standardized cryptographic constructs well. It will not try to support proposed standards, or cryptographic constructs which exist only in academia. Cryptographic standards which are settled upon concurrently with or after the RISC-V vector cryptographic extensions standardization will be dealt with by future RISC-V vector cryptographic standard extensions.
- Historically, there has been some discussion (Lee et al., 2004) on how newly supported operations in general-purpose computing might enable new bases for cryptographic algorithms. The standard will not try to anticipate new useful low-level operations which *may* be useful as building blocks for future cryptographic constructs.
- Regarding side-channel countermeasures: Where relevant, proposed instructions must aim to remove the possibility of any timing side-channels. All instructions shall be implemented with data-independent timing. That is, the latency of the execution of these instructions shall not vary with

different input values.

33.1.4. Element Groups

Many vector crypto instructions operate on operands that are wider than elements (which are currently limited to 64 bits wide). Typically, these operands are 128- and 256-bits wide. In many cases, these operands are comprised of smaller operands that are combined (for example, each SHA-2 operand is comprised of 4 words). However, in other cases these operands are a single value (for example, in the AES round instructions, each operand is 128-bit block or round key).

We treat these operands as a vector of one or more *element groups* as defined in Section 31.19.

Each vector crypto instruction that operates on element groups explicitly specifies their three defining parameters: EGW, EGS, and EEW.

Instruction Group	Extension	EGW	EEW	EGS
AES	Zvkned	128	32	4
SHA256	zvknh[ab]	128	32	4
SHA512	zvknhb	256	64	4
GCM	Zvkg	128	32	4
SM4	Zvksed	128	32	4
SM3	Zvksh	256	32	8

- *Element Group Width (`EGW`) - total number of bits in an element group*
- *Effective Element Width (`EEW`) - number of bits in each element*
- *Element Group Size (`EGS`) - number of elements in an element group*

For all of the vector crypto instructions in this specification, `EEW`=`SEW`.

The required `SEW` for each cryptographic instruction was chosen to match what is typically needed for other instructions when implementing the targeted algorithm.

- A **Vector Element Group** is a vector of one or more element groups.
- A **Scalar Element Group** is a single element group.

Element groups can be formed across registers in implementations where `VLEN`< `EGW` by using an `LMUL`>1.

Since the **vector extension for application processors** *requires a minimum of VLEN of 128, at most such implementations would require LMUL=2 to form the largest element groups in this specification.*

However, implementations with a smaller VLEN, such as embedded designs, will requires a larger `LMUL` to form the necessary element groups. It is important to keep in mind that this reduces the number of register groups available such that it may be difficult or impossible to write efficient code for the intended cryptographic algorithms.

For example, an implementation with `VLEN`=32 would need to set `LMUL`=8 to create a 256-bit element group for `SM3`. This would mean that there would only be 4 register groups, 3 of which would be consumed by a single `SM3` message-expansion instruction.

As with all vector instructions, the number of elements processed is specified by the vector length `vl`. The

number of element groups operated upon is then `vl`/`EGS`. Likewise the starting element group is `vstart`/`EGS`. See Section 33.1.5 for limitations on `vl` and `vstart` for vector crypto instructions.

33.1.5. Instruction Constraints

All standard vector instruction constraints specified by RVV 1.0 apply to Vector Crypto instructions. In addition to those constraints a few additional specific constraints are introduced.

The following is a quick reference for the various constraints of specific Vector Crypto instructions.

vl and vstart constraints

Since `vl` and `vstart` refer to elements, Vector Crypto instructions that use elements groups (See Section 33.1.4) require that these values are an integer multiple of the Element Group Size (`EGS`).

- Instructions that violate the `vl` or `vstart` requirements are *reserved*.

Instructions	EGS
vaes*	4
vsha2*	4
vg*	4
vsm3*	8
vsm4*	4

LMUL constraints

For element-group instructions, `LMUL`*`VLEN` must always be at least as large as `EGW`, otherwise an *illegal-instruction exception* is raised, even if `vl`=0.

Instructions	SEW	EGW
vaes*	32	128
vsha2*	32	128
vsha2*	64	256
vg*	32	128
vsm3*	32	256
vsm4*	32	128

SEW constraints

Some Vector Crypto instructions are only defined for a specific `SEW`. In such a case all other `SEW` values are *reserved*.

Instructions	Required SEW
vaes*	32
Zvknha: vsha2*	32
Zvknhb: vsha2*	32 or 64
vclmul[h]	64
vg*	32
vsm3*	32
vsm4*	32

Vector/Scalar constraints

This specification defines new vector/scalar (.vs) instructions that uses **Scalar Element Groups**. The **Scalar Element Group** operand has `EMUL = ceil(EGW / VLEN)`.

Scalar element group operands do not need to be aligned to LMUL for any implementation with VLEN >= EGW.

In the case of the `.vs` instructions defined in this specification, `vs2` holds a 128-bit scalar element group. For implementations with `VLEN` ≥ 128, `vs2` refers to a single register. Thus, the `vd` register group must not overlap the `vs2` register. However, in implementations where `VLEN` < 128, `vs2` refers to a register group comprised of the number of registers needed to hold the 128-bit scalar element group. In this case, the `vd` register group must not overlap this `vs2` register group.

Instruction	Register	Cannot Overlap
vaes*.vs	vs2	vd
vsm4r.vs	vs2	vd
vsha2c[hl]	vs1, vs2	vd
vsha2ms	vs1, vs2	vd
vsm3me	vs2	vd
vsm3c	vs2	vd

33.1.6. Vector-Scalar Instructions

The RISC-V Vector Extension defines three encodings for Vector-Scalar operations which get their scalar operand from a GPR or FP register:

- OPIVX: Scalar GPR *x* register
- OPFVF: Scalar FP *f* register
- OPMVX: Scalar GPR *x* register

However, the Vector Extensions include Vector Reduction Operations which can also be considered Vector-Scalar operations because a scalar operand is provided from element 0 of vector register `vs1`. The vector operand is provided in vector register group `vs2`. These reduction operations all use the `.vs` suffix in their mnemonics. Additionally, the reduction operations all produce a scalar result in element 0 of the destination register, `vd`.

The Vector Crypto Extensions define Vector-Scalar instructions that are similar to these Vector Reduction Operations in that they get a scalar operand from a vector register. However, they differ in that they get a scalar element group (see Section 33.1.4) from `vs2` and they return *vector* results to `vd`, which is also a source vector operand. These Vector-Scalar crypto instructions also use the `.vs` suffix in their mnemonics.

We chose to use `vs2` as the scalar operand, and `vd` as the vector operand, so that we could use the `vs1` specifier as additional encoding bits for these instructions. This allows these instructions to have a much smaller encoding footprint, leaving more rooms for other instructions in the future.

These instructions enable a single key, specified as a scalar element group in `vs2`, to be applied to each element group of register group `vd`.

Scalar element groups will occupy at most a single register in application processors. However, in implementations where VLEN<128, they will occupy 2 (VLEN=64) or 4 (VLEN=32) registers.

It is common for multiple AES encryption rounds (for example) to be performed in parallel with the same round key (e.g. in counter modes). Rather than having to first splat the common key across the whole vector group, these vector-scalar crypto instructions allow the round key to be specified as a scalar element group.

33.1.7. Software Portability

The following contains some guidelines that enable the portability of vector-crypto-based code to implementations with different values for `VLEN`

Application Processors

Application processors are expected to follow the V-extension and will therefore have `VLEN` ≥ 128.

Since most of the *cryptography-specific* instructions have an `EGW`=128, nothing special needs to be done for these instructions to support implementations with `VLEN`=128.

However, the SHA-512 and SM3 instructions have an `EGW`=256. Implementations with `VLEN` = 128, require that `LMUL` is doubled for these instructions in order to create 256-bit elements across a pair of registers. Code written with this doubling of `LMUL` will not affect the results returned by implementations with `VLEN` ≥ 256 because `vl` controls how many element groups are processed. Therefore, we recommend that libraries that implement SHA-512 and SM3 employ this doubling of `LMUL` to ensure that the software can run on all implementation with `VLEN` ≥ 128.

While the doubling of `LMUL` for these instructions is *safe* for implementations with `VLEN` ≥ 256, it may be less optimal as it will result in unnecessary register pressure and might exact a performance penalty in some microarchitectures. Therefore, we suggest that in addition to providing portable code for SHA-512 and SM3, libraries should also include more optimal code for these instructions when `VLEN` ≥ 256.

Algorithm	Instructions	VLEN	LMUL
SHA-512	vsha2*	64	vl/2
SM3	vsm3*	32	vl/4

Embedded Processors

Embedded processors will typically have implementations with `VLEN` < 128. This will require code to be written with larger `LMUL` values to enable the element groups to be formed.

The `.vs` instructions require scalar element groups of `EGW`=128. On implementations with `VLEN` < 128, these scalar element groups will necessarily be formed across registers. This is different from most scalars in vector instructions that typically consume part of a single register.

We recommend that different code be available for `VLEN`=32 and `VLEN`=64, as code written for `VLEN`=32 will likely be too burdensome for `VLEN`=64 implementations.

33.2. Extensions Overview

The section introduces all of the extensions in the Vector Cryptography Instruction Set Extension Specification.

The Zvknhb and Zvbc Vector Crypto Extensions --and accordingly the composite extensions Zvkn, Zvknc, Zvkng, and Zvksc-- depend on Zve64x.

All of the other Vector Crypto Extensions depend on `Zve32x`.

Note: If `Zve32x` is supported then `Zvkb` or `Zvbb` provide support for EEW of 8, 16, and 32. If `Zve64x` is supported then `Zvkb` or `Zvbb` also add support for EEW 64.

All *cryptography-specific* instructions defined in this Vector Crypto specification (i.e., those in Zvkned, Zvknh[ab], Zvkg, Zvksed and Zvksh but *not* Zvbb,Zvkb, or Zvbc) shall be executed with data-independent execution latency as defined in the RISC-V Scalar Cryptography Extensions specification. It is important to note that the Vector Crypto instructions are independent of the implementation of the `Zkt` extension and do not require that `Zkt` is implemented.

This specification includes a Zvkt extension that, when implemented, requires certain vector instructions (including Zvbb, Zvkb, and Zvbc) to be executed with data-independent execution latency.

Detection of individual cryptography extensions uses the unified software-based RISC-V discovery method.

At the time of writing, these discovery mechanisms are still a work in progress.

33.2.1. `Zvbb` - Vector Basic Bit-manipulation

Vector basic bit-manipulation instructions.

This extension is a superset of the Zvkb extension.

Mnemonic	Instruction
vandn.[vv,vx]	Vector And-Not
vbrev.v	Vector Reverse Bits in Elements
vbrev8.v	Vector Reverse Bits in Bytes
vrev8.v	Vector Reverse Bytes
vclz.v	Vector Count Leading Zeros
vctz.v	Vector Count Trailing Zeros
vcpop.v	Vector Population Count
vrol.[vv,vx]	Vector Rotate Left
vror.[vv,vx,vi]	Vector Rotate Right
vwsll.[vv,vx,vi]	Vector Widening Shift Left Logical

33.2.2. `Zvbc` - Vector Carry-less Multiplication

General purpose carry-less multiplication instructions which are commonly used in cryptography and hashing (e.g., Elliptic curve cryptography, GHASH, CRC).

These instructions are only defined for `SEW`=64.

Mnemonic	Instruction
vclmul.[vv,vx]	Vector Carry-less Multiply
vclmulh.[vv,vx]	Vector Carry-less Multiply Return High Half

33.2.3. Zvkb - Vector Cryptography Bit-manipulation

Vector bit-manipulation instructions that are essential for implementing common cryptographic workloads securely & efficiently.

This Zvkb extension is a proper subset of the Zvbb extension. Zvkb allows for vector crypto implementations without incurring the cost of implementing the additional bitmanip instructions in the Zvbb extension: vbrev.v, vclz.v, vctz.v, vcpop.v, and vwsll.[vv,vx,vi].

Mnemonic	Instruction
vandn.[vv,vx]	Vector And-Not
vbrev8.v	Vector Reverse Bits in Bytes
vrev8.v	Vector Reverse Bytes
vrol.[vv,vx]	Vector Rotate Left
vror.[vv,vx,vi]	Vector Rotate Right

33.2.4. `Zvkg` - Vector GCM/GMAC

Instructions to enable the efficient implementation of $GHASH_H$ which is used in Galois/Counter Mode (GCM) and Galois Message Authentication Code (GMAC).

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

$GHASH_H$ is defined in the "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC" (Dworkin, 2007) (NIST Specification).

GCM is used in conjunction with block ciphers (e.g., AES and SM4) to encrypt a message and provide authentication. GMAC is used to provide authentication of a message without encryption.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is `vl`/`EGS`. `vl` must be set to the number of `SEW=32` elements to be processed and therefore must be a multiple of `EGS=4`.
Likewise, `vstart` must be a multiple of `EGS=4`.

SEW	EGW	Mnemonic	Instruction
32	128	vghsh.vv	Vector GHASH Add-Multiply
32	128	vgmul.vv	Vector GHASH Multiply

33.2.5. `Zvkned` - NIST Suite: Vector AES Block Cipher

Instructions for accelerating encryption, decryption and key-schedule functions of the AES block cipher as defined in Federal Information Processing Standards Publication 197 (NIST, 2001)

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

For the best performance, it is suggested that these instruction be implemented on systems with `VLEN`>=128. On systems with `VLEN`<128, element groups may be formed by concatenating 32-bit elements from two or four registers by using an LMUL =2 and LMUL=4 respectively.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is `vl`/`EGS`. `vl` must be set to the number of `SEW=32` elements to be processed and therefore must be a multiple of `EGS=4`.
Likewise, `vstart` must be a multiple of `EGS=4`.

SEW	EGW	Mnemonic	Instruction
32	128	vaesef.[vv,vs]	Vector AES encrypt final round
32	128	vaesem.[vv,vs]	Vector AES encrypt middle round
32	128	vaesdf.[vv,vs]	Vector AES decrypt final round
32	128	vaesdm.[vv,vs]	Vector AES decrypt middle round
32	128	vaeskf1.vi	Vector AES-128 Forward KeySchedule
32	128	vaeskf2.vi	Vector AES-256 Forward KeySchedule
32	128	vaesz.vs	Vector AES round zero

33.2.6. `Zvknh[ab]` - NIST Suite: Vector SHA-2 Secure Hash

Instructions for accelerating SHA-2 as defined in FIPS PUB 180-4 Secure Hash Standard (SHS) (NIST, 2015)

`SEW` differentiates between SHA-256 (`SEW`=32) and SHA-512 (`SEW`=64).

- SHA-256: these instructions work on 128-bit element groups comprised of four 32-bit elements.
- SHA-512: these instructions work on 256-bit element groups comprised of four 64-bit elements.

SEW	EGW	SHA-2	Extension
32	128	SHA-256	Zvknha, Zvknhb
64	256	SHA-512	Zvknhb

- Zvknhb supports SHA-256 and SHA-512.
- Zvknha supports only SHA-256.

SHA-256 implementations with VLEN < 128 require LMUL>1 to combine 32-bit elements from register groups to provide all four elements of the element group.

SHA-512 implementations with VLEN < 256 require LMUL>1 to combine 64-bit elements from register groups to provide all four elements of the element group.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is `vl`/`EGS`. `vl` must be set to the number of `SEW` elements to be processed and therefore must be a multiple of `EGS=4`.
Likewise, `vstart` must be a multiple of `EGS=4`.

Mnemonic	Instruction
vsha2ms.vv	Vector SHA-2 Message Schedule
vsha2c[hl].vv	Vector SHA-2 Compression

33.2.7. `Zvksed` - ShangMi Suite: SM4 Block Cipher

Instructions for accelerating encryption, decryption, and key-schedule functions of the SM4 block cipher.

The SM4 block cipher is specified in *32907-2016: {SM4} Block Cipher Algorithm* (*GB/T 32907-2016: SM4 Block Cipher Algorithm, 2016*)

There are other various sources available that describe the SM4 block cipher. While not the final version of the standard, RFC 8998 ShangMi (SM) Cipher Suites for TLS 1.3 is useful and easy to access.

All of these instructions work on 128-bit element groups comprised of four 32-bit elements.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is `vl`/`EGS`. `vl` must be set to the number of `SEW=32` elements to be processed and therefore must be a multiple of `EGS=4`.
Likewise, `vstart` must be a multiple of `EGS=4`.

SEW	EGW	Mnemonic	Instruction
32	128	vsm4k.vi	Vector SM4 Key Expansion
32	128	vsm4r.[vv,vs]	SM4 Block Cipher Rounds

33.2.8. `Zvksh` - ShangMi Suite: SM3 Secure Hash

Instructions for accelerating functions of the SM3 Hash Function.

The SM3 secure hash algorithm is specified in *32905-2016: SM3 Cryptographic Hash Algorithm* (*GB/T 32905-2016: SM3 Cryptographic Hash Algorithm*, 2016)

There are other various sources available that describe the SM3 secure hash. While not the final version of the standard, RFC 8998 ShangMi (SM) Cipher Suites for TLS 1.3 is useful and easy to access.

All of these instructions work on 256-bit element groups comprised of eight 32-bit elements.

Implementations with VLEN < 256 require LMUL>1 to combine 32-bit elements from register groups to provide all eight elements of the element group.

To help avoid side-channel timing attacks, these instructions shall be implemented with data-independent timing.

The number of element groups to be processed is `vl`/`EGS`. `vl` must be set to the number of `SEW=32` elements to be processed and therefore must be a multiple of `EGS=8`.
Likewise, `vstart` must be a multiple of `EGS=8`.

SEW	EGW	Mnemonic	Instruction
32	256	vsm3me.vv	SM3 Message Expansion
32	256	vsm3c.vi	SM3 Compression

33.2.9. Zvkn - NIST Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkned	Zvkned
Zvknhb	Zvknhb
Zvkb	Zvkb
Zvkt	Zvkt

While Zvkg and Zvbc are not part of this extension, it is recommended that at least one of them is implemented with this extension to enable efficient AES-GCM.

33.2.10. Zvknc - NIST Algorithm Suite with carry-less multiply

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkn	Zvkn
Zvbc	Zvbc

This extension combines the NIST Algorithm Suite with the vector carry-less multiply extension to enable AES-GCM.

33.2.11. `Zvkng` - NIST Algorithm Suite with GCM

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvkn	Zvkn
Zvkg	Zvkg

This extension combines the NIST Algorithm Suite with the GCM/GMAC extension to enable high-performance AES-GCM.

33.2.12. Zvks - ShangMi Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvksed	Zvksed
Zvksh	Zvksh
Zvkb	Zvkb
Zvkt	Zvkt

While Zvkg and Zvbc are not part of this extension, it is recommended that at least one of them is implemented with this extension to enable efficient SM4-GCM.

33.2.13. Zvksc - ShangMi Algorithm Suite with carry-less multiplication

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvks	Zvks
Zvbc	Zvbc

This extension combines the ShangMi Algorithm Suite with the vector carry-less multiply extension to enable SM4-GCM.

33.2.14. Zvksg - ShangMi Algorithm Suite with GCM

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Zvks	Zvks
Zvkg	Zvkg

This extension combines the ShangMi Algorithm Suite with the GCM/GMAC extension to enable high-performance SM4-GCM.

33.2.15. `Zvkt` - Vector Data-Independent Execution Latency

The Zvkt extension requires all implemented instructions from the following list to be executed with data-independent execution latency as defined in the RISC-V Scalar Cryptography Extensions specification.

Data-independent execution latency (DIEL) applies to all *data operands* of an instruction, even those that are not a part of the body or that are inactive. However, DIEL does not apply to other values such as vl, vtype, and the mask (when used to control execution of a masked vector instruction). Also, DIEL does not apply to constant values specified in the instruction encoding such as the use of the zero register (`x0`), and, in the case of immediate forms of an instruction, the values in the immediate fields (i.e., imm, and uimm).

In some cases --- which are explicitly specified in the lists below --- operands that are used as control rather than data are exempt from DIEL.

DIEL helps protect against side-channel timing attacks that are used to determine data values that are intended to be kept secret. Such values include cryptographic keys, plain text, and partially encrypted text. DIEL is not intended to keep software (and cryptographic algorithms contained therein) secret as it is assumed that an adversary would already know these. This is why DIEL doesn't apply to constants embedded in instruction encodings.

It is important that the values of elements that are not in the body or that are masked off do not affect the execution latency of the instruction. Sometimes such elements contain data that also needs to be kept secret.

33.2.15.1. All Zvbb instructions

- vandn.v[vx]
- vclz.v
- vcpop.v
- vctz.v
- vbrev.v
- vbrev8.v
- vrev8.v
- vrol.v[vx]
- vror.v[vxi]
- vwsll.[vv,vx,vi]

All Zvkb instructions are also covered by DIEL as they are a proper subset of Zvbb

33.2.15.2. All Zvbc instructions

- vclmul[h].v[vx]

33.2.15.3. add/sub

- v[r]sub.v[vx]
- vadd.v[ivx]
- vsub.v[vx]

- vwadd[u].[vw][vx]
- vwsub[u].[vw][vx]

33.2.15.4. add/sub with carry

- vadc.v[ivx]m
- vmadc.v[ivx][m]
- vmsbc.v[vx]m
- vsbc.v[vx]m

33.2.15.5. compare and set

- vmseq.v[vxi]
- vmsgt[u].v[xi]
- vmsle[u].v[xi]
- vmslt[u].v[xi]
- vmsne.v[ivx]

33.2.15.6. copy

- vmv.s.x
- vmv.v.[ivxs]
- vmv[1248]r.v

33.2.15.7. extend

- vsext.vf[248]
- vzext.vf[248]

33.2.15.8. logical

- vand.v[ivx]
- vm[n]or.mm
- vmand[n].mm
- vmnand.mm
- vmorn.mm
- vmx[n]or.mm
- vor.v[ivx]
- vxor.v[ivx]

33.2.15.9. multiply

- vmul[h].v[vx]

- vmulh[s]u.v[vx]
- vwmul.v[vx]
- vwmul[s]u.v[vx]

33.2.15.10. multiply-add

- vmacc.v[vx]
- vmadd.v[vx]
- vnmsac.v[vx]
- vnmsub.v[vx]
- vwmacc.v[vx]
- vwmacc[s]u.v[vx]
- vwmaccus.vx

33.2.15.11. Integer Merge

- vmerge.v[ivx]m

33.2.15.12. permute

In the `.vv` and `.xv` forms of the `vrgather[ei16]` instructions, the values in `vs1` and `rs1` are used for control and therefore are exempt from DIEL.

- vrgather.v[ivx]
- vrgatherei16.vv

33.2.15.13. shift

- vnsr[al].w[ivx]
- vsll.v[ivx]
- vsr[al].v[ivx]

33.2.15.14. slide

- vslide1[up|down].vx
- vfslide1[up|down].vf

In the vslide[up|down].vx instructions, the value in `rs1` is used for control (i.e., slide amount) and therefore is exempt from DIEL.

- vslide[up|down].v[ix]

The following instructions are not affected by Zvkt:

- All storage operations
- All floating-point operations

- *add/sub saturate*
  - *vsadd[u].v[ivx]*
  - *vssub[u].v[vx]*
- *clip*
  - *vnclip[u].w[ivx]*
- *compress*
  - *vcompress.vm*
- *divide*
  - *vdiv[u].v[vx]*
  - *vrem[u].v[vx]*
- *average*
  - *vaadd[u].v[vx]*
  - *vasub[u].v[vx]*
- *mask Op*
  - *vcpop.m*
  - *vfirst.m*
  - *vid.v*
  - *viota.m*
  - *vms[bio]f.m*
- *min/max*
  - *vmax[u].v[vx]*
  - *vmin[u].v[vx]*
- *Multiply-saturate*
  - *vsmul.v[vx]*
- *reduce*
  - *vredsum.vs*
  - *vwredsum[u].vs*
  - *vred[and|or|xor].vs*
  - *vred[min|max][u].vs*
- *shift round*
  - *vssra.v[ivx]*
  - *vssrl.v[ivx]*
- *vset*
  - *vsetivli*
  - *vsetvl[i]*

33.3. Instructions

33.3.1. vaesdf.[vv,vs]

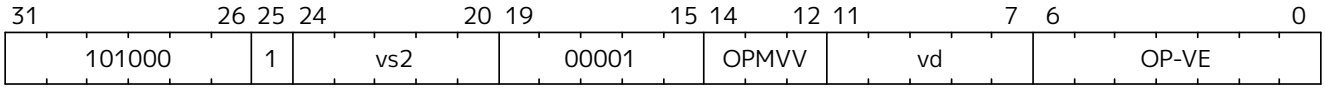
Synopsis

Vector AES final-round decryption

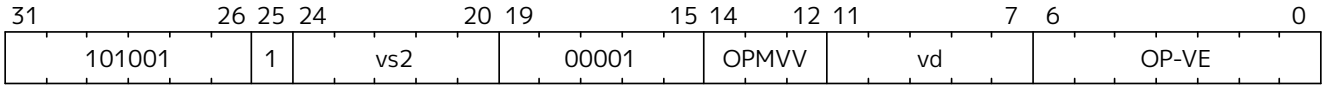
Mnemonic

vaesdf.vv vd, vs2
vaesdf.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- Only for the `.vs` form: the `vd` register group overlaps the `vs2` scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	round key
Vd	output	128	4	32	new round state

Description

A final-round AES block cipher decryption is performed.

The InvShiftRows and InvSubBytes steps are applied to each round state element group from `vd`. This is then XORed with the round key in either the corresponding element group in `vs2` (vector-vector form) or scalar element group in `vs2` (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESDF(vs2, vd, suffix)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {
```

```
  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
    let keyelem = if suffix == "vv" then i else 0;
    let state : bits(128) = get_velem(vd,  EGW=128, i);
    let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
    let sr    : bits(128) = aes_shift_rows_inv(state);
    let sb    : bits(128) = aes_subbytes_inv(sr);
    let ark   : bits(128) = sb ^ rkey;
    set_velem(vd, EGW=128, i, ark);
  }
  RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.2. vaesdm.[vv,vs]

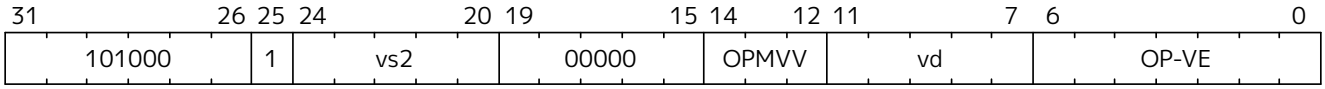
Synopsis

Vector AES middle-round decryption

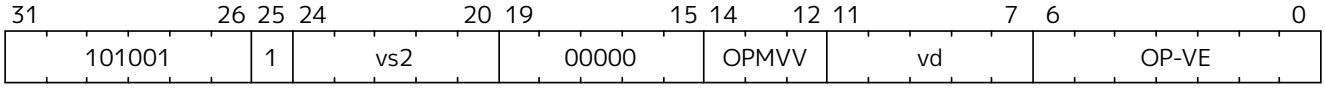
Mnemonic

vaesdm.vv vd, vs2
vaesdm.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- Only for the `.vs` form: the `vd` register group overlaps the `vs2` scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	round key
Vd	output	128	4	32	new round state

Description

A middle-round AES block cipher decryption is performed.

The InvShiftRows and InvSubBytes steps are applied to each round state element group from `vd`. This is then XORed with the round key in either the corresponding element group in `vs2` (vector-vector form) or the scalar element group in `vs2` (vector-scalar form). The result is then applied to the InvMixColumns step.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESDM(vs2, vd, suffix)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)
```

```
  foreach (i from eg_start to eg_len-1) {
    let keyelem = if suffix == "vv" then i else 0;
    let state : bits(128) = get_velem(vd, EGW=128, i);
    let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
    let sr    : bits(128) = aes_shift_rows_inv(state);
    let sb    : bits(128) = aes_subbytes_inv(sr);
    let ark   : bits(128) = sb ^ rkey;
    let mix   : bits(128) = aes_mixcolumns_inv(ark);
    set_velem(vd, EGW=128, i, mix);
  }
  RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.3. vaesef.[vv,vs]

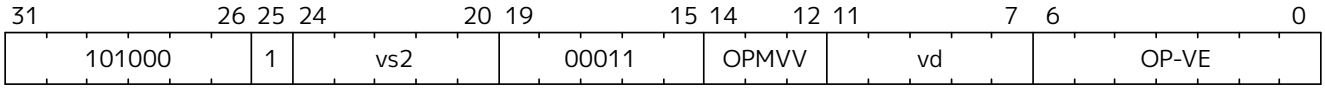
Synopsis

Vector AES final-round encryption

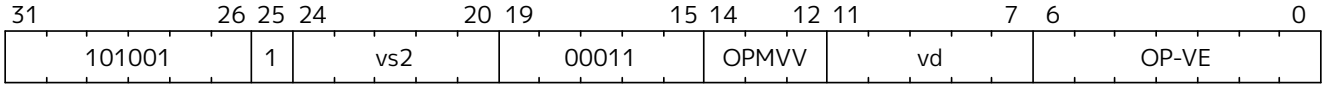
Mnemonic

vaesef.vv vd, vs2
vaesef.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- Only for the `.vs` form: the `vd` register group overlaps the `vs2` scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
vd	input	128	4	32	round state
vs2	input	128	4	32	round key
vd	output	128	4	32	new round state

Description

A final-round encryption function of the AES block cipher is performed.

The SubBytes and ShiftRows steps are applied to each round state element group from `vd`. This is then XORed with the round key in either the corresponding element group in `vs2` (vector-vector form) or the scalar element group in `vs2` (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESEF(vs2, vd, suffix) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)
```

```
  foreach (i from eg_start to eg_len-1) {
    let keyelem = if suffix == "vv" then i else 0;
    let state : bits(128) = get_velem(vd, EGW=128, i);
    let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
    let sb    : bits(128) = aes_subbytes_fwd(state);
    let sr    : bits(128) = aes_shift_rows_fwd(sb);
    let ark   : bits(128) = sr ^ rkey;
    set_velem(vd, EGW=128, i, ark);
  }
  RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.4. vaesem.[vv,vs]

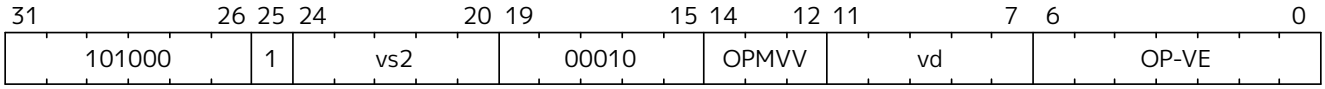
Synopsis

Vector AES middle-round encryption

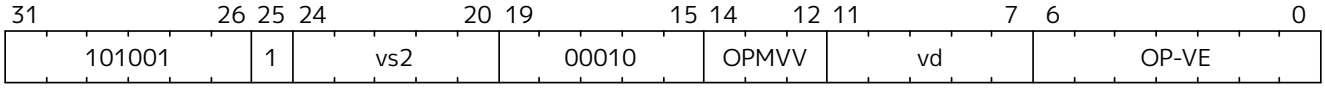
Mnemonic

vaesem.vv vd, vs2
vaesem.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- Only for the `.vs` form: the `vd` register group overlaps the `vs2` scalar element group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	round state
Vs2	input	128	4	32	Round key
Vd	output	128	4	32	new round state

Description

A middle-round encryption function of the AES block cipher is performed.

The SubBytes, ShiftRows, and MixColumns steps are applied to each round state element group from `vd`. This is then XORed with the round key in either the corresponding element group in `vs2` (vector-vector form) or the scalar element group in `vs2` (vector-scalar form).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

Operation

```
function clause execute (VAESEM(vs2, vd, suffix)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)
```

```
  foreach (i from eg_start to eg_len-1) {
    let keyelem = if suffix == "vv" then i else 0;
    let state : bits(128) = get_velem(vd, EGW=128, i);
    let rkey  : bits(128) = get_velem(vs2, EGW=128, keyelem);
    let sb    : bits(128) = aes_subbytes_fwd(state);
    let sr    : bits(128) = aes_shift_rows_fwd(sb);
    let mix   : bits(128) = aes_mixcolumns_fwd(sr);
    let ark   : bits(128) = mix ^ rkey;
    set_velem(vd, EGW=128, i, ark);
  }
  RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.5. vaeskf1.vi

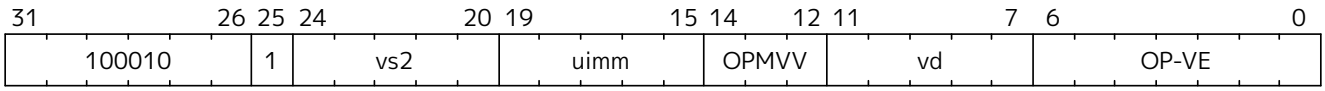
Synopsis

Vector AES-128 Forward KeySchedule generation

Mnemonic

vaeskf1.vi vd, vs2, uimm

Encoding



Reserved Encodings

- `SEW` is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
uimm	input	-	-	-	Round Number (rnd)
Vs2	input	128	4	32	Current round key
Vd	output	128	4	32	Next round key

Description

A single round of the forward AES-128 KeySchedule is performed.

The next round key is generated word by word from the current round key element group in `vs2` and the immediately previous word of the round key. The least significant word is generated using the most significant word of the current round key as well as a round constant which is selected by the round number.

The round number, which ranges from 1 to 10, comes from `uimm[3:0]`; `uimm[4]` is ignored. The out-of-range `uimm[3:0]` values of 0 and 11-15 are mapped to in-range values by inverting `uimm[3]`. Thus, 0 maps to 8, and 11-15 maps to 3-7. The round number is used to specify a round constant which is used in generating the first round key word.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

We chose to map out-of-range round numbers to in-range values as this allows the instruction's behavior to be fully defined for all values of `uimm[4:0]` *with minimal extra logic.*

Operation

```
function clause execute (VAESKF1(rnd, vd, vs2)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

 // project out-of-range immediates onto in-range values
 if( (unsigned(rnd[3:0]) > 10) | (rnd[3:0] = 0)) then rnd[3] = ~rnd[3]
```

```
  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  let r : bits(4) = rnd-1;

  foreach (i from eg_start to eg_len-1) {
      let CurrentRoundKey[3:0]  : bits(128)  = get_velem(vs2, EGW=128, i);
      let w[0] : bits(32) = aes_subword_fwd(aes_rotword(CurrentRoundKey[3]))
XOR
        aes_decode_rcon(r) XOR CurrentRoundKey[0]
      let w[1] : bits(32) = w[0] XOR CurrentRoundKey[1]
      let w[2] : bits(32) = w[1] XOR CurrentRoundKey[2]
      let w[3] : bits(32) = w[2] XOR CurrentRoundKey[3]
      set_velem(vd, EGW=128, i, w[3:0]);
    }
    RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.6. vaeskf2.vi

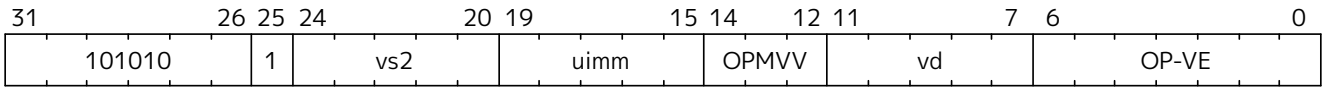
Synopsis

Vector AES-256 Forward KeySchedule generation

Mnemonic

vaeskf2.vi vd, vs2, uimm

Encoding



Reserved Encodings

- `SEW` is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	Previous Round key
uimm	input	-	-	-	Round Number (rnd)
Vs2	input	128	4	32	Current Round key
Vd	output	128	4	32	Next round key

Description

A single round of the forward AES-256 KeySchedule is performed.

The next round key is generated word by word from the previous round key element group in `vd` and the immediately previous word of the round key. The least significant word of the next round key is generated by applying a function to the most significant word of the current round key and then XORing the result with the round constant. The round number is used to select the round constant as well as the function.

The round number, which ranges from 2 to 14, comes from `uimm[3:0]`; `uimm[4]` is ignored. The out-of-range `uimm[3:0]` values of 0-1 and 15 are mapped to in-range values by inverting `uimm[3]`. Thus, 0-1 maps to 8-9, and 15 maps to 7.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

We chose to map out-of-range round numbers to in-range values as this allows the instruction's behavior to be fully defined for all values of `uimm[4:0]` *with minimal extra logic.*

Operation

```
function clause execute (VAESKF2(rnd, vd, vs2)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

 // project out-of-range immediates into in-range values
 if((unsigned(rnd[3:0]) < 2) |  (unsigned(rnd[3:0]) > 14)) then rnd[3] =
```

```
~rnd[3]

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
      let CurrentRoundKey[3:0]  : bits(128)  = get_velem(vs2, EGW=128, i);
      let RoundKeyB[3:0] : bits(128)  = get_velem(vd, EGW=128, i); // Previous
round key

      let w[0] : bits(32) = if (rnd[0]==1) then
        aes_subword_fwd(CurrentRoundKey[3]) XOR RoundKeyB[0];
      else
        aes_subword_fwd(aes_rotword(CurrentRoundKey[3])) XOR
aes_decode_rcon((rnd>>1) - 1) XOR RoundKeyB[0];
      w[1] : bits(32) = w[0] XOR RoundKeyB[1]
      w[2] : bits(32) = w[1] XOR RoundKeyB[2]
      w[3] : bits(32) = w[2] XOR RoundKeyB[3]
      set_velem(vd, EGW=128, i, w[3:0]);
    }
    RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.7. vaesz.vs

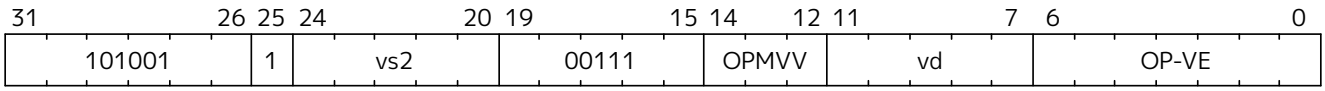
Synopsis

Vector AES round zero encryption/decryption

Mnemonic

vaesz.vs vd, vs2

Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- The `vd` register group overlaps the `vs2` register

Arguments

Register	Direction	EGW	EGS	EEW	Definition
vd	input	128	4	32	round state
vs2	input	128	4	32	round key
vd	output	128	4	32	new round state

Description

A round-0 AES block cipher operation is performed. This operation is used for both encryption and decryption.

There is only a `.vs` form of the instruction. `Vs2` holds a scalar element group that is used as the round key for all of the round state element groups. The new round state output of each element group is produced by XORing the round key with each element group of `vd`.

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

This instruction is needed to avoid the need to "splat" a 128-bit vector register group when the round key is the same for all 128-bit "lanes". Such a splat would typically be implemented with a `vrgather` *instruction which would hurt performance in many implementations. This instruction only exists in the* `.vs` *form because the* `.vv` *form would be identical to the* `vxor.vv vd, vs2, vd` *instruction.*

Operation

```
function clause execute (VAESZ(vs2, vd) = {
  if(((vstart%EGS)<>0) | (LMUL*VLEN < EGW))  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)
```

```
  foreach (i from eg_start to eg_len-1) {
    let state : bits(128) = get_velem(vd, EGW=128, i);
    let rkey  : bits(128) = get_velem(vs2, EGW=128, 0);
    let ark   : bits(128) = state ^ rkey;
    set_velem(vd, EGW=128, i, ark);
  }
  RETIRE_SUCCESS
  }
}
```

Included in

Zvkn, Zvknc, Zvkned, Zvkng

33.3.8. vandn.[vv,vx]

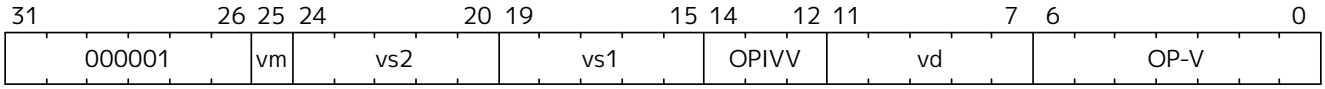
Synopsis

Bitwise And-Not

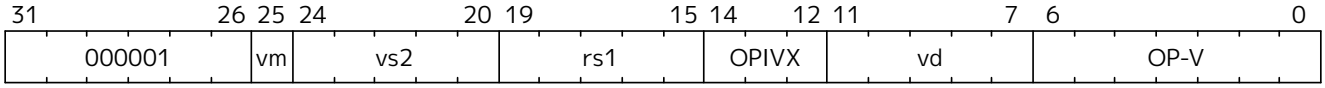
Mnemonic

vandn.vv vd, vs2, vs1, vm
vandn.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Op1 (to be inverted)
Vs2	input	Op2
Vd	output	Result

Vector-Scalar Arguments

Register	Direction	Definition
Rs1	input	Op1 (to be inverted)
Vs2	input	Op2
Vd	output	Result

Description

A bitwise *and-not* operation is performed.

Each bit of `Op1` is inverted and logically ANDed with the corresponding bits in `vs2`. In the vector-scalar version, `Op1` is the sign-extended or truncated value in scalar register `rs1`. In the vector-vector version, `Op1` is `vs1`.

Note on necessity of instruction

This instruction is performance-critical to SHA3. Specifically, the Chi step of the FIPS 202 Keccak Permutation. Emulating it via 2 instructions is expected to have significant performance impact. The `.vv` *form of the instruction is what is needed for SHA3; the* `.vx` *form was added for completeness.*

There is no .vi version of this instruction because the same functionality can be achieved by using an inversion of the immediate value with the `vand.vi` *instruction.*

Operation

```
function clause execute (VANDN(vs2, vs1, vd, suffix)) = {
  foreach (i from vstart to vl-1) {
    let op1 = match suffix {
      "vv" => get_velem(vs1, SEW, i),
      "vx" => sext_or_truncate_to_sew(X(vs1))
    };
    let op2 = get_velem(vs2, SEW, i);
    set_velem(vd, EEW=SEW, i, ~op1 & op2);
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb, Zvkb, Zvkn, Zvknc, Zvkng, Zvks Zvksc, Zvksg

33.3.9. vbrev.v

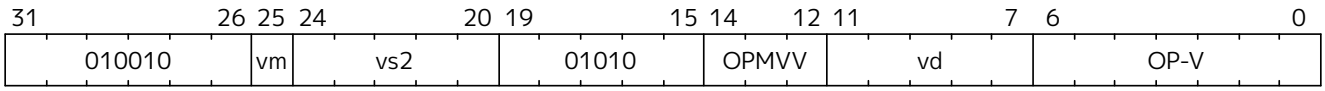
Synopsis

Vector Reverse Bits in Elements

Mnemonic

vbrev.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Elements with bits reversed

Description

A bit reversal is performed on the bits of each element.

Operation

```
function clause execute (VBREV(vs2)) = {

  foreach (i from vstart to vl-1) {
    let input = get_velem(vs2, SEW, i);
    let output : bits(SEW) = 0;
    foreach (i from 0 to SEW-1)
      let output[SEW-1-i] = input[i];
    set_velem(vd, SEW, i, output)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb

33.3.10. vbrev8.v

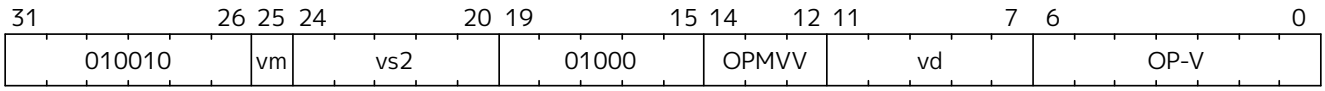
Synopsis

Vector Reverse Bits in Bytes

Mnemonic

vbrev8.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Elements with bit-reversed bytes

Description

A bit reversal is performed on the bits of each byte.

This instruction is commonly used for GCM when the zvkg extension is not implemented. This byte-wise instruction is defined for all SEWs to eliminate the need to change SEW when operating on wider elements.

Operation

```
function clause execute (VBREV8(vs2)) = {

  foreach (i from vstart to vl-1) {
    let input = get_velem(vs2, SEW, i);
    let output : bits(SEW) = 0;
    foreach (i from 0 to SEW-8 by 8)
      let output[i+7..i] = reverse_bits_in_byte(input[i+7..i]);
    set_velem(vd, SEW, i, output)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb, Zvkb, Zvkn, Zvknc, Zvkng, Zvks Zvksc, Zvksg

33.3.11. vclmul.[vv,vx]

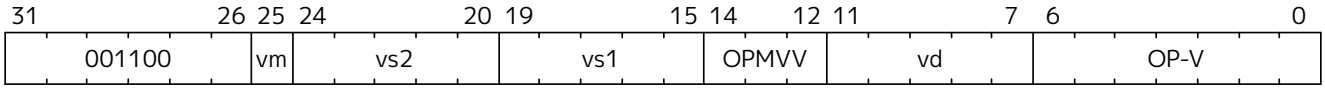
Synopsis

Vector Carry-less Multiply by vector or scalar - returning low half of product.

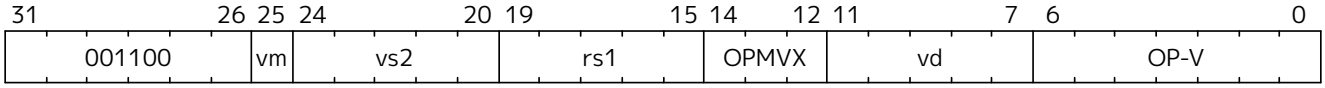
Mnemonic

vclmul.vv vd, vs2, vs1, vm
vclmul.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 64

Arguments

Register	Direction	Definition
Vs1/Rs1	input	multiplier
Vs2	input	multiplicand
Vd	output	carry-less product low

Description

Produces the low half of 128-bit carry-less product.

Each 64-bit element in the `vs2` vector register is carry-less multiplied by either each 64-bit element in `vs1` (vector-vector), or the 64-bit value from integer register `rs1` (vector-scalar). The result is the least significant 64 bits of the carry-less product.

The 64-bit carry-less multiply instructions can be used for implementing GCM in the absence of the `zvkg` extension. We do not make these instructions exclusive as the 64-bit carry-less multiply is readily derived from the instructions in the `zvkg` extension and can have utility in other areas. Likewise, we treat other SEW values as reserved so as not to preclude future extensions from using this opcode with different element widths. For example, a future extension might define an `SEW`=32 version of this instruction to enable `Zve32*` implementations to have vector carry-less multiplication instructions.

Operation

```
function clause execute (VCLMUL(vs2, vs1, vd, suffix)) = {

  foreach (i from vstart to vl-1) {
    let op1 : bits (64) = if suffix =="vv" then get_velem(vs1,i)
                          else zext_or_truncate_to_sew(X(vs1));
    let op2 : bits (64) = get_velem(vs2,i);
```

```
    let product : bits (64) = clmul(op1,op2,SEW);
    set_velem(vd, i, product);
  }
  RETIRE_SUCCESS
}

function clmul(x, y, width) = {
  let result : bits(width) = zeros();
  foreach (i from 0 to (width - 1)) {
    if y[i] == 1 then result = result ^ (x << i);
  }
  result
}
```

Included in

Zvbc, Zvknc, Zvksc

33.3.12. vclmulh.[vv,vx]

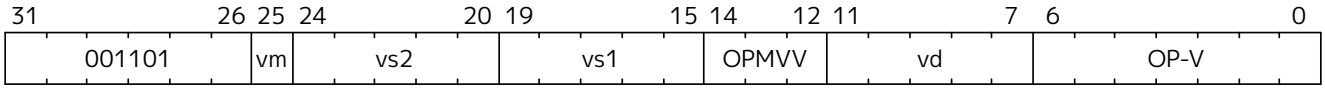
Synopsis

Vector Carry-less Multiply by vector or scalar - returning high half of product.

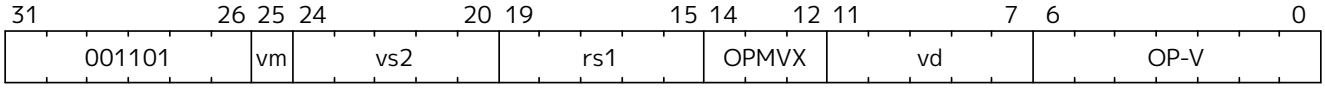
Mnemonic

vclmulh.vv vd, vs2, vs1, vm
vclmulh.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 64

Arguments

Register	Direction	Definition
Vs1	input	multiplier
Vs2	input	multiplicand
Vd	output	carry-less product high

Description

Produces the high half of 128-bit carry-less product.

Each 64-bit element in the `vs2` vector register is carry-less multiplied by either each 64-bit element in `vs1` (vector-vector), or the 64-bit value from integer register `rs1` (vector-scalar). The result is the most significant 64 bits of the carry-less product.

Operation

```
function clause execute (VCLMULH(vs2, vs1, vd, suffix)) = {

  foreach (i from vstart to vl-1) {
    let op1 : bits (64) = if suffix =="vv" then get_velem(vs1,i)
                          else zext_or_truncate_to_sew(X(vs1));
    let op2 : bits (64) = get_velem(vs2, i);
    let product : bits (64) = clmulh(op1, op2, SEW);
    set_velem(vd, i, product);
  }
  RETIRE_SUCCESS
}

function clmulh(x, y, width) = {
```

```
  let result : bits(width) = 0;
  foreach (i from 1 to (width - 1)) {
    if y[i] == 1 then result = result ^ (x >> (width - i));
  }
  result
}
```

Included in

Zvbc, Zvknc, Zvksc

33.3.13. vclz.v

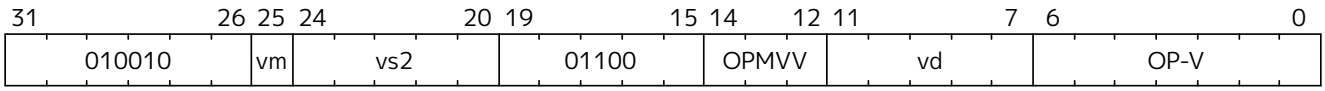
Synopsis

Vector Count Leading Zeros

Mnemonic

vclz.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of leading zero bits

Description

A leading zero count is performed on each element.

The result for zero-valued inputs is the value SEW.

Operation

```
function clause execute (VCLZ(vs2)) = {

  foreach (i from vstart to vl-1) {
    let input = get_velem(vs2, SEW, i);
    for (j = (SEW - 1); j >= 0;  j--)
      if [input[j]] == 0b1 then break;
    set_velem(vd, SEW, i, SEW - 1 - j)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb

33.3.14. vcpop.v

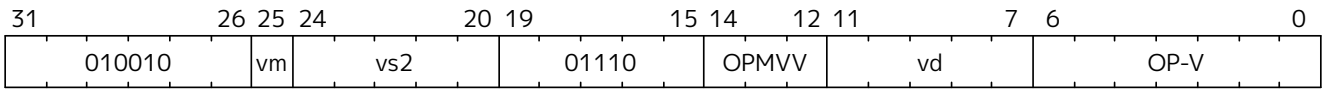
Synopsis

Count the number of bits set in each element

Mnemonic

vcpop.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of bits set

Description

A population count is performed on each element.

Operation

```
function clause execute (VCPOP(vs2)) = {

  foreach (i from vstart to vl-1) {
    let input = get_velem(vs2, SEW, i);
    let output : bits(SEW) = 0;
    for (j = 0; j < SEW;  j++)
      output = output + input[j];
    set_velem(vd, SEW, i, output)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb

33.3.15. vctz.v

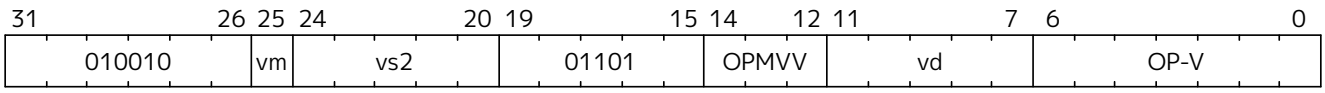
Synopsis

Vector Count Trailing Zeros

Mnemonic

vctz.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Count of trailing zero bits

Description

A trailing zero count is performed on each element.

Operation

```
function clause execute (VCTZ(vs2)) = {

  foreach (i from vstart to vl-1) {
    let input = get_velem(vs2, SEW, i);
    for (j = 0; j < SEW;  j++)
      if [input[j]] == 0b1 then break;
    set_velem(vd, SEW, i, j)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb

33.3.16. vghsh.vv

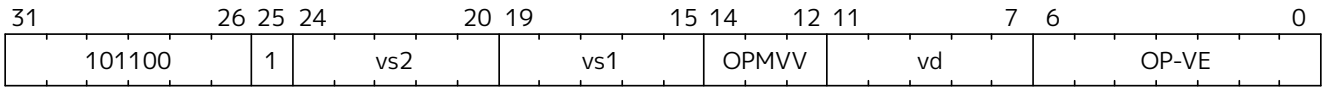
Synopsis

Vector Add-Multiply over GHASH Galois-Field

Mnemonic

vghsh.vv vd, vs2, vs1

Encoding



Reserved Encodings

- `SEW` is any value other than 32

Arguments

Register	Direction	EGW	EGS	SEW	Definition
Vd	input	128	4	32	Partial hash (Y_i)
Vs1	input	128	4	32	Cipher text (X_i)
Vs2	input	128	4	32	Hash Subkey (H)
Vd	output	128	4	32	Partial-hash (Y_{i+1})

Description

A single "iteration" of the $GHASH_H$ algorithm is performed.

This instruction treats all of the inputs and outputs as 128-bit polynomials and performs operations over GF[2]. It produces the next partial hash (Y_{i+1}) by adding the current partial hash (Y_i) to the cipher text block (X_i) and then multiplying (over GF(2^{128})) this sum by the Hash Subkey (H).

The multiplication over GF(2^{128}) is a carry-less multiply of two 128-bit polynomials modulo GHASH's irreducible polynomial ($x^{128} + x^7 + x^2 + x + 1$).

The operation can be compactly defined as $Y_{i+1} = ((Y_i \wedge X_i) \cdot H)$

The NIST specification (see Zvkg) orders the coefficients from left to right $x_0x_1x_2...x_{127}$ for a polynomial $x_0 + x_1u + x_2 u^2 + ... + x_{127}u^{127}$. This can be viewed as a collection of byte elements in memory with the byte containing the lowest coefficients (i.e., 0,1,2,3,4,5,6,7) residing at the lowest memory address. Since the bits in the bytes are reversed, this instruction internally performs bit swaps within bytes to put the bits in the standard ordering (e.g., 7,6,5,4,3,2,1,0).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

We are bit-reversing the bytes of inputs and outputs so that the intermediate values are consistent with the NIST specification. These reversals are inexpensive to implement as they unconditionally swap bit positions and therefore do not require any logic.

Since the same hash subkey `H` will typically be used repeatedly on a given message, a future extension might define a vector-scalar version of this instruction where `vs2` is the scalar element group. This would help reduce register pressure when `LMUL` > 1.

Operation

```
function clause execute (VGHSH(vs2, vs1, vd)) = {
  // operands are input with bits reversed in each byte
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
    let Y = get_velem(vd,EGW=128,i);  // current partial-hash
    let X = get_velem(vs1,EGW=128,i);  // block cipher output
    let H = brev8(get_velem(vs2,EGW=128,i)); // Hash subkey

    let Z : bits(128) = 0;

    let S = brev8(Y ^ X);

    for (int bit = 0; bit < 128; bit++) {
      if bit_to_bool(S[bit])
        Z ^= H

      bool reduce = bit_to_bool(H[127]);
      H = H << 1; // left shift H by 1
      if (reduce)
        H ^= 0x87; // Reduce using x^7 + x^2 + x^1 + 1 polynomial
    }

    let result = brev8(Z); // bit reverse bytes to get back to GCM standard
ordering
    set_velem(vd, EGW=128, i, result);
  }
  RETIRE_SUCCESS
 }
}
```

Included in

Zvkg, Zvkng, Zvksg

33.3.17. vgmul.vv

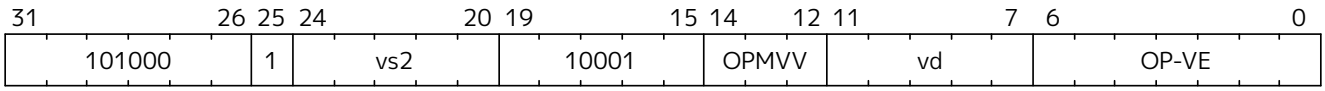
Synopsis

Vector Multiply over GHASH Galois-Field

Mnemonic

vgmul.vv vd, vs2

Encoding



Reserved Encodings

- `SEW` is any value other than 32

Arguments

Register	Direction	EGW	EGS	SEW	Definition
Vd	input	128	4	32	Multiplier
Vs2	input	128	4	32	Multiplicand
Vd	output	128	4	32	Product

Description

A $GHASH_H$ multiply is performed.

This instruction treats all of the inputs and outputs as 128-bit polynomials and performs operations over GF[2]. It produces the product over GF(2^{128}) of the two 128-bit inputs.

The multiplication over GF(2^{128}) is a carry-less multiply of two 128-bit polynomials modulo GHASH's irreducible polynomial ($x^{128} + x^7 + x^2 + x + 1$).

The NIST specification (see Zvkg) orders the coefficients from left to right $x_0x_1x_2...x_{127}$ for a polynomial $x_0 + x_1u + x_2 u^2 + ... + x_{127}u^{127}$. This can be viewed as a collection of byte elements in memory with the byte containing the lowest coefficients (i.e., 0,1,2,3,4,5,6,7) residing at the lowest memory address. Since the bits in the bytes are reversed, This instruction internally performs bit swaps within bytes to put the bits in the standard ordering (e.g., 7,6,5,4,3,2,1,0).

This instruction must always be implemented such that its execution latency does not depend on the data being operated upon.

We are bit-reversing the bytes of inputs and outputs so that the intermediate values are consistent with the NIST specification. These reversals are inexpensive to implement as they unconditionally swap bit positions and therefore do not require any logic.

Since the same multiplicand will typically be used repeatedly on a given message, a future extension might define a vector-scalar version of this instruction where `vs2` *is the scalar element group. This would help reduce register pressure when* `LMUL` *> 1.*

This instruction is identical to `vghsh.vv` *with vs1=0. This instruction is often used in GHASH code. In some cases it is followed by an XOR to perform a multiply-add. Implementations may choose to fuse these two instructions to improve performance on GHASH code that doesn't use the add-multiply form of the* `vghsh.vv` *instruction.*

Operation

```
function clause execute (VGMUL(vs2, vs1, vd)) = {
  // operands are input with bits reversed in each byte
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
    let Y = brev8(get_velem(vd,EGW=128,i));  // Multiplier
    let H = brev8(get_velem(vs2,EGW=128,i)); // Multiplicand
    let Z : bits(128) = 0;

    for (int bit = 0; bit < 128; bit++) {
      if bit_to_bool(Y[bit])
        Z ^= H

      bool reduce = bit_to_bool(H[127]);
      H = H << 1; // left shift H by 1
      if (reduce)
        H ^= 0x87; // Reduce using x^7 + x^2 + x^1 + 1 polynomial
    }


    let result = brev8(Z);
    set_velem(vd, EGW=128, i, result);
  }
  RETIRE_SUCCESS
 }
}
```

Included in

Zvkg, Zvkng, Zvksg

33.3.18. vrev8.v

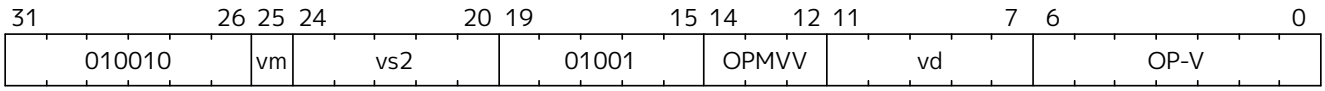
Synopsis

Vector Reverse Bytes

Mnemonic

vrev8.v vd, vs2, vm

Encoding (Vector)



Arguments

Register	Direction	Definition
Vs2	input	Input elements
Vd	output	Byte-reversed elements

Description

A byte reversal is performed on each element of `vs2`, effectively performing an endian swap.

This element-wise endian swapping is needed for several cryptographic algorithms including SHA2 and SM3.

Operation

```
function clause execute (VREV8(vs2)) = {
  foreach (i from vstart to vl-1) {
    input = get_velem(vs2, SEW, i);
    let output : SEW = 0;
    let j = SEW - 1;
    foreach (k from 0 to (SEW - 8) by 8) {
      output[k..(k + 7)] = input[(j - 7)..j];
      j = j - 8;
    set_velem(vd, SEW, i, output)
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb, Zvkb, Zvkn, Zvknc, Zvkng, Zvks Zvksc, Zvksg

33.3.19. vrol.[vv,vx]

Synopsis

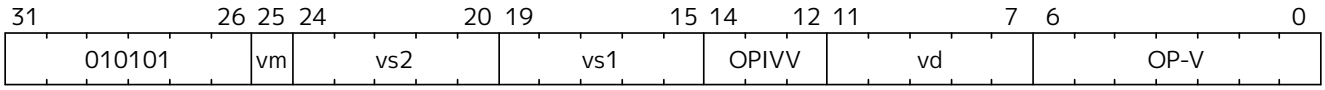
Vector rotate left by vector/scalar.

Mnemonic

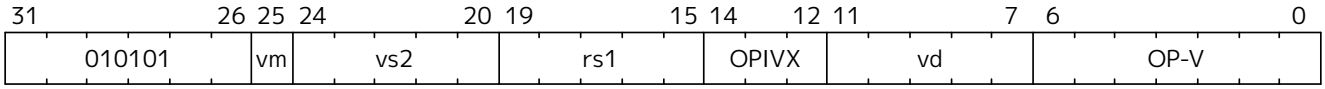
vrol.vv vd, vs2, vs1, vm

vrol.vx vd, vs2, rs1, vm

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Vector-Scalar Arguments

Register	Direction	Definition
Rs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Description

A bitwise left rotation is performed on each element of `vs2`

The elements in `vs2` are rotated left by the rotate amount specified by either the corresponding elements of `vs1` (vector-vector), or integer register `rs1` (vector-scalar). Only the low log2(`SEW`) bits of the rotate-amount value are used, all other bits are ignored.

There is no immediate form of this instruction (i.e., `vrol.vi`) as the same result can be achieved by negating the rotate amount and using the immediate form of rotate right instruction (i.e., vror.vi).

Operation

```
function clause execute (VROL_VV(vs2, vs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) <<< (get_velem(vs1, i) & (SEW-1))
```

```
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VROL_VX(vs2, rs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) <<< (X(rs1) & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb, Zvkb, Zvkn, Zvknc, Zvkng, Zvks Zvksc, Zvksg

33.3.20. vror.[vv,vx,vi]

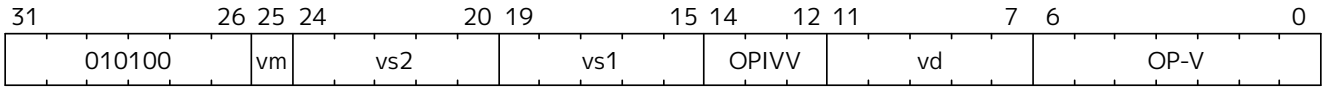
Synopsis

Vector rotate right by vector/scalar/immediate.

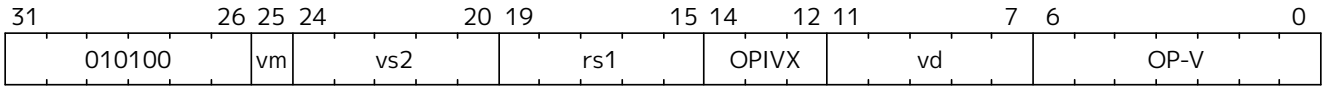
Mnemonic

vror.vv vd, vs2, vs1, vm
vror.vx vd, vs2, rs1, vm
vror.vi vd, vs2, uimm, vm

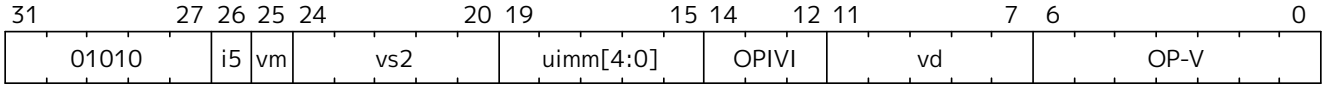
Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Encoding (Vector-Immediate)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Vector-Scalar/Immediate Arguments

Register	Direction	Definition
Rs1/imm	input	Rotate amount
Vs2	input	Data
Vd	output	Rotated data

Description

A bitwise right rotation is performed on each element of `vs2`.

The elements in `vs2` are rotated right by the rotate amount specified by either the corresponding elements of `vs1` (vector-vector), integer register `rs1` (vector-scalar), or an immediate value (vector-immediate). Only the low log2(`SEW`) bits of the rotate-amount value are used, all other bits are ignored.

Operation

```
function clause execute (VROR_VV(vs2, vs1, vd)) = {
  foreach (i from vstart to vl - 1) {
```

```
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (get_velem(vs1, i) & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VROR_VX(vs2, rs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (X(rs1) & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VROR_VI(vs2, uimm[5:0], vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=SEW, i,
      get_velem(vs2, i) >>> (uimm[5:0] & (SEW-1))
    )
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb, Zvkb, Zvkn, Zvknc, Zvkng, Zvks Zvksc, Zvksg

33.3.21. vsha2c[hl].vv

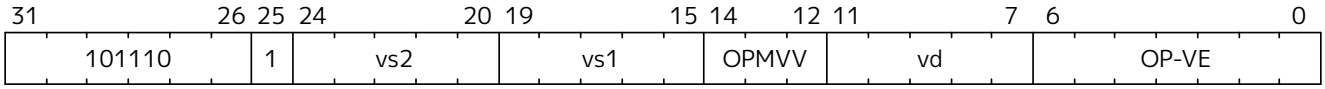
Synopsis

Vector SHA-2 two rounds of compression.

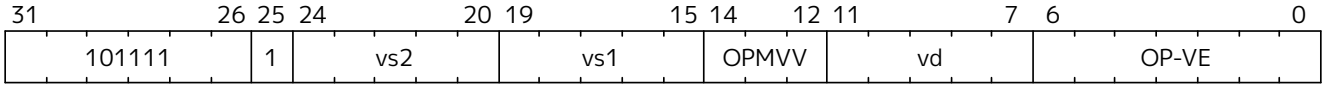
Mnemonic

vsha2ch.vv vd, vs2, vs1
vsha2cl.vv vd, vs2, vs1

Encoding (Vector-Vector) High part



Encoding (Vector-Vector) Low part



Reserved Encodings

- `zvknha`: `SEW` is any value other than 32
- `zvknhb`: `SEW` is any value other than 32 or 64
- The `vd` register group overlaps with either `vs1` or `vs2`

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	4*SEW	4	SEW	current state {c, d, g, h}
Vs1	input	4*SEW	4	SEW	MessageSched plus constant[3:0]
Vs2	input	4*SEW	4	SEW	current state {a, b, e, f}
Vd	output	4*SEW	4	SEW	next state {a, b, e, f}

Description

- `SEW`=32: 2 rounds of SHA-256 compression are performed (`zvknha` and `zvknhb`)
- `SEW`=64: 2 rounds of SHA-512 compression are performed (`zvknhb`)

Two words of `vs1` are processed with the 8 words of current state held in `vd` and `vs2` to perform two rounds of hash computation producing four words of the next state.

Note to software developers

The NIST standard (see zvknh[ab]) requires the final hash to be in big-endian byte ordering within SEW-sized words. Since this instruction treats all words as little-endian, software needs to perform an endian swap on the final output of this instruction after all of the message blocks have been processed.

The `vsha2ch` version of this instruction uses the two most significant message schedule words from the element group in `vs1` while the `vsha2cl` version uses the two least significant message schedule words. Otherwise, these versions of the instruction are identical. Having a high and low version of this instruction typically improves performance when interleaving independent hashing operations (i.e., when hashing several files at once).

Preventing overlap between `vd` *and* `vs1` *or* `vs2` *simplifies implementation with* `VLEN < EGW`*. This restriction does not have any coding impact since proper implementation of the algorithm requires that* `vd`*,* `vs1` *and* `vs2` *each are different registers.*

Operation

```
function clause execute (VSHA2c(vs2, vs1, vd)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
   let {a @ b @ e @ f} : bits(4*SEW) = get_velem(vs2, 4*SEW, i);
   let {c @ d @ g @ h} : bits(4*SEW) = get_velem(vd, 4*SEW, i);
   let MessageSchedPlusC[3:0] : bits(4*SEW) = get_velem(vs1, 4*SEW, i);
   let {W1, W0} == VSHA2cl ? MessageSchedPlusC[1:0] : MessageSchedPlusC[3:2];
// l vs h difference is the words selected

   let T1 : bits(SEW) = h + sum1(e) + ch(e,f,g) + W0;
   let T2 : bits(SEW) = sum0(a) + maj(a,b,c);
   h  = g;
   g  = f;
   f  = e;
   e  = d + T1;
   d  = c;
   c  = b;
   b  = a;
   a  = T1 + T2;


   T1  = h + sum1(e) + ch(e,f,g) + W1;
   T2  = sum0(a) + maj(a,b,c);
   h = g;
   g = f;
   f = e;
   e = d + T1;
   d = c;
   c = b;
   b = a;
   a = T1 + T2;
   set_velem(vd, 4*SEW, i, {a @ b @ e @ f});
  }
  RETIRE_SUCCESS
  }
}
```

```
function sum0(x) = {
 match SEW {
  32 => rotr(x,2)  XOR rotr(x,13) XOR rotr(x,22),
  64 => rotr(x,28) XOR rotr(x,34) XOR rotr(x,39)
 }
}

function sum1(x) = {
 match SEW {
  32 => rotr(x,6)  XOR rotr(x,11) XOR rotr(x,25),
  64 => rotr(x,14) XOR rotr(x,18) XOR rotr(x,41)
 }
}

function ch(x, y, z) = ((x & y) ^ ((~x) & z))


function maj(x, y, z) =  ((x & y) ^ (x & z) ^ (y & z))

function ROTR(x,n) = (x >> n) | (x << SEW - n)
```

Included in

Zvkn, Zvknc, Zvkng, zvknh[ab]

33.3.22. vsha2ms.vv

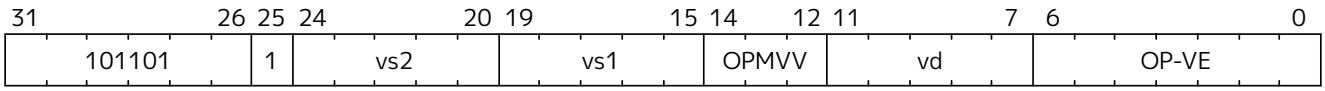
Synopsis

Vector SHA-2 message schedule.

Mnemonic

vsha2ms.vv vd, vs2, vs1

Encoding (Vector-Vector)



Reserved Encodings

- `zvknha`: `SEW` is any value other than 32
- `zvknhb`: `SEW` is any value other than 32 or 64
- The `vd` register group overlaps with either `vs1` or `vs2`

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	4*SEW	4	SEW	Message words {W[3], W[2], W[1], W[0]}
Vs2	input	4*SEW	4	SEW	Message words {W[11], W[10], W[9], W[4]}
Vs1	input	4*SEW	4	SEW	Message words {W[15], W[14], -, W[12]}
Vd	output	4*SEW	4	SEW	Message words {W[19], W[18], W[17], W[16]}

Description

- `SEW`=32: Four rounds of SHA-256 message schedule expansion are performed (`zvknha` and `zvknhb`)
- `SEW`=64: Four rounds of SHA-512 message schedule expansion are performed (`zvknhb`)

Eleven of the last 16 `SEW`-sized message-schedule words from `vd` (oldest), `vs2`, and `vs1` (most recent) are processed to produce the next 4 message-schedule words.

Note to software developers

The first 16 SEW-sized words of the message schedule come from the message block in big-endian byte order. Since this instruction treats all words as little endian, software is required to endian swap these words.

All of the subsequent message schedule words are produced by this instruction and therefore do not require an endian swap.

Note to software developers

Software is required to pack the words into element groups as shown above in the arguments table. The indices indicate the relate age with lower indices indicating older words.

Note to software developers

The {W_{11}, W_{10}, W_9, W_4} element group can easily be formed by using a vector vmerge instruction with the appropriate mask (for example with `vl=4` and `4b0001` as the 4 mask bits)

```
vmerge.vvm {W11, W10, W9, W4}, {W11, W10, W9, W8}, {W7, W6, W5, W4}, V0
```

Preventing overlap between `vd` *and* `vs1` *or* `vs2` *simplifies implementation with* `VLEN < EGW`*. This restriction does not have any coding impact since proper implementation of the algorithm requires that* `vd`*,* `vs1` *and* `vs2` *each contain different portions of the message schedule.*

Operation

```
function clause execute (VSHA2ms(vs2, vs1, vd)) = {
  // SEW32 = SHA-256
  // SEW64 =  SHA-512
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
    {W[3] @  W[2] @  W[1] @  W[0]}  : bits(EGW) = get_velem(vd, EGW, i);
    {W[11] @ W[10] @ W[9] @  W[4]}  : bits(EGW) = get_velem(vs2, EGW, i);
    {W[15] @ W[14] @ W[13] @ W[12]} : bits(EGW) = get_velem(vs1, EGW, i);

    W[16] = sig1(W[14]) + W[9]  + sig0(W[1]) + W[0];
    W[17] = sig1(W[15]) + W[10] + sig0(W[2]) + W[1];
    W[18] = sig1(W[16]) + W[11] + sig0(W[3]) + W[2];
    W[19] = sig1(W[17]) + W[12] + sig0(W[4]) + W[3];

    set_velem(vd, EGW, i, {W[19] @ W[18] @ W[17] @ W[16]});
  }
  RETIRE_SUCCESS
  }
}

function sig0(x) = {
 match SEW {
  32 => (ROTR(x,7) XOR ROTR(x,18) XOR SHR(x,3)),
  64 => (ROTR(x,1) XOR ROTR(x,8) XOR SHR(x,7)));
 }
}

function sig1(x) = {
 match SEW {
  32 => (ROTR(x,17) XOR ROTR(x,19) XOR SHR(x,10),
  64 => ROTR(x,19) XOR ROTR(x,61) XOR SHR(x,6));
 }
}

function ROTR(x,n) = (x >> n) | (x << SEW - n)
function SHR (x,n) = x >> n
```

Included in

Zvkn, Zvknc, Zvkng, zvknh[ab]

33.3.23. vsm3c.vi

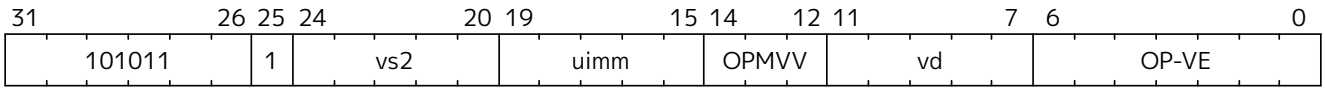
Synopsis

Vector SM3 Compression

Mnemonic

vsm3c.vi vd, vs2, uimm

Encoding



Reserved Encodings

- `SEW` is any value other than 32
- The `vd` register group overlaps with the `vs2` register group

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	256	8	32	Current state {H,G.F,E,D,C,B,A}
uimm	input	-	-	-	round number (rnds)
Vs2	input	256	8	32	Message words {-,-,w[5],w[4],-,-,w[1],w[0]}
Vd	output	256	8	32	Next state {H,G.F,E,D,C,B,A}

Description

Two rounds of SM3 compression are performed.

The current state of eight 32-bit words is read in as an element group from `vd`. Eight 32-bit message words are read in as an element group from `vs2`, although only four of them are used. All of the 32-bit input words are byte-swapped from big endian to little endian. These inputs are processed somewhat differently based on the round group (as specified in rnds), and the next state is generated as an element group of eight 32-bit words. The next state of eight 32-bit words are generated, swapped from little endian to big endian, and are returned in an eight-element group.

The round number is provided by the 5-bit `rnds` unsigned immediate. Legal values are 0 - 31 and indicate which group of two rounds are being performed. For example, if rnds=1, then rounds 2 and 3 are being performed.

The round number is used in the rotation of the constant as well to inform the behavior which differs between rounds 0-15 and rounds 16-63.

The endian byte swapping of the input and output words enables us to align with the SM3 specification without requiring that software perform these swaps.

Preventing overlap between `vd` and `vs2` simplifies implementation with `VLEN < EGW`. This restriction does not have any coding impact since proper implementation of the algorithm requires that `vd` and `vs2` each are different registers.

Operation

```
function clause execute (VSM3C(rnds, vs2, vd)) = {
```

```
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {

  // load state
  let {Hi @ Gi @ Fi @ Ei @ Di @ Ci @ Bi @ Ai} : bits(256) : bits(256) =
(get_velem(vd, 256, i));
  //load message schedule
  let {u_w7 @ u_w6 @ w5i @ w4i @ u_w3 @ u_w2 @ w1i @ w0i} : bits(256) =
(get_velem(vs2, 256, i));
  // u_w inputs are unused

// perform endian swap
let H : bits(32) = rev8(Hi);
let G : bits(32) = rev8(Gi);
let F : bits(32) = rev8(Fi);
let E : bits(32) = rev8(Ei);
let D : bits(32) = rev8(Di);
let C : bits(32) = rev8(Ci);
let B : bits(32) = rev8(Bi);
let A : bits(32) = rev8(Ai);

let w5 = : bits(32) rev8(w5i);
let w4 = : bits(32) rev8(w4i);
let w1 = : bits(32) rev8(w1i);
let w0 = : bits(32) rev8(w0i);

let x0 :bits(32) = w0 ^ w4;  // W'[0]
let x1 :bits(32) = w1 ^ w5;  // W'[1]

let j = 2 * rnds;
let ss1 : bits(32) = ROL32(ROL32(A, 12) + E + ROL32(T_j(j), j % 32), 7);
let ss2 : bits(32) = ss1 ^ ROL32(A, 12);
let tt1 : bits(32) = FF_j(A, B, C, j) + D + ss2 + x0;
let tt2 : bits(32) = GG_j(E, F, G, j) + H + ss1 + w0;
D = C;
let : bits(32) C1 = ROL32(B, 9);
B = A;
let A1 : bits(32) = tt1;
H = G;
let G1 : bits(32) = ROL32(F, 19);
F = E;
let E1 : bits(32) = P_0(tt2);
```

```
j = 2 * rnds + 1;
ss1 = ROL32(ROL32(A1, 12) + E1 + ROL32(T_j(j), j % 32), 7);
ss2 = ss1 ^ ROL32(A1, 12);
tt1 = FF_j(A1, B, C1, j) + D + ss2 + x1;
tt2 = GG_j(E1, F, G1, j) + H + ss1 + w1;
D = C1;
let C2 : bits(32) = ROL32(B, 9);
B = A1;
let A2 : bits(32) = tt1;
H = G1;
let G2 = : bits(32) ROL32(F, 19);
F = E1;
let E2 = : bits(32) P_0(tt2);

// Update the destination register - swap back to big endian
let result : bits(256) = {rev8(G1) @ rev8(G2) @ rev8(E1) @ rev8(E2) @ rev8(C1)
@ rev8(C2) @ rev8(A1) @ rev8(A2)};
set_velem(vd, 256, i, result);
      }

RETIRE_SUCCESS
  }
}

function FF1(X, Y, Z) = ((X) ^ (Y) ^ (Z))
function FF2(X, Y, Z) = (((X) & (Y)) | ((X) & (Z)) | ((Y) & (Z)))

function FF_j(X, Y, Z, J) = (((J) <= 15) ? FF1(X, Y, Z) : FF2(X, Y, Z))

function GG1(X, Y, Z) = ((X) ^ (Y) ^ (Z))
function GG2(X, Y, Z) = (((X) & (Y)) | ((~(X)) & (Z)))
.
function GG_j(X, Y, Z, J) = (((J) <= 15) ? GG1(X, Y, Z) : GG2(X, Y, Z))

function T_j(J) = (((J) <= 15) ? (0x79CC4519) : (0x7A879D8A))

function P_0(X) = ((X) ^ ROL32((X),  9) ^ ROL32((X), 17))
```

Included in

Zvks, Zvksc, Zvksg, Zvksh

33.3.24. vsm3me.vv

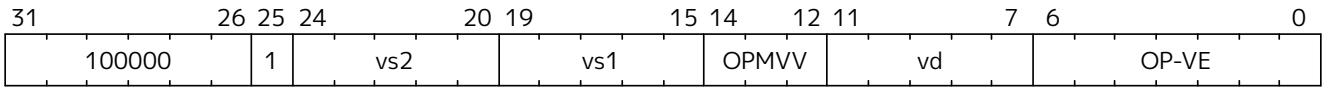
Synopsis

Vector SM3 Message Expansion

Mnemonic

vsm3me.vv vd, vs2, vs1

Encoding



Reserved Encodings

- `SEW` is any value other than 32
- The `vd` register group overlaps with the `vs2` register group.

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vs1	input	256	8	32	Message words W[7:0]
Vs2	input	256	8	32	Message words W[15:8]
Vd	output	256	8	32	Message words W[23:16]

Description

Eight rounds of SM3 message expansion are performed.

The sixteen most recent 32-bit message words are read in as two eight-element groups from `vs1` and `vs2`. Each of these words is swapped from big endian to little endian. The next eight 32-bit message words are generated, swapped from little endian to big endian, and are returned in an eight-element group.

The endian byte swapping of the input and output words enables us to align with the SM3 specification without requiring that software perform these swaps.

Preventing overlap between `vd` and `vs2` simplifies implementations with `VLEN < EGW`. This restriction should not have any coding impact since the algorithm requires these values to be preserved for generating the next 8 words.

Operation

```
function clause execute (VSM3ME(vs2, vs1)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  foreach (i from eg_start to eg_len-1) {
    let w[7:0]  : bits(256) = get_velem(vs1, 256, i);
```

```
    let w[15:8] : bits(256) = get_velem(vs2, 256, i);

    // Byte Swap inputs from big-endian to little-endian
    let w15 = rev8(w[15]);
    let w14 = rev8(w[14]);
    let w13 = rev8(w[13]);
    let w12 = rev8(w[12]);
    let w11 = rev8(w[11]);
    let w10 = rev8(w[10]);
    let w9  = rev8(w[9]);
    let w8  = rev8(w[8]);
    let w7  = rev8(w[7]);
    let w6  = rev8(w[6]);
    let w5  = rev8(w[5]);
    let w4  = rev8(w[4]);
    let w3  = rev8(w[3]);
    let w2  = rev8(w[2]);
    let w1  = rev8(w[1]);
    let w0  = rev8(w[0]);

    // Note that some of the newly computed words are used in later
invocations.
    let w[16] = ZVKSH_W(w0 @  w7 @  w13 @   w3 @  w10);
    let w[17] = ZVKSH_W(w1 @  w8 @  w14 @   w4 @  w11);
    let w[18] = ZVKSH_W(w2 @  w9 @  w15 @   w5 @  w12);
    let w[19] = ZVKSH_W(w3 @ w10 @  w16 @   w6 @  w13);
    let w[20] = ZVKSH_W(w4 @ w11 @  w17 @   w7 @  w14);
    let w[21] = ZVKSH_W(w5 @ w12 @  w18 @   w8 @  w15);
    let w[22] = ZVKSH_W(w6 @ w13 @  w19 @   w9 @  w16);
    let w[23] = ZVKSH_W(w7 @ w14 @  w20 @  w10 @  w17);

  // Byte swap outputs from little-endian back to big-endian
    let w16 : Bits(32) = rev8(W[16]);
    let w17 : Bits(32) = rev8(W[17]);
    let w18 : Bits(32) = rev8(W[18]);
    let w19 : Bits(32) = rev8(W[19]);
    let w20 : Bits(32) = rev8(W[20]);
    let w21 : Bits(32) = rev8(W[21]);
    let w22 : Bits(32) = rev8(W[22]);
    let w23 : Bits(32) = rev8(W[23]);


    // Update the destination register.
    set_velem(vd, 256, i, {w23 @ w22 @ w21 @ w20 @ w19 @ w18 @ w17 @ w16});
  }
  RETIRE_SUCCESS
  }
}

  function P_1(X) ((X) ^ ROL32((X), 15) ^ ROL32((X), 23))
```

```
function ZVKSH_W(M16, M9, M3, M13, M6) = \
(P1( (M16) ^  (M9) ^ ROL32((M3), 15) ) ^ ROL32((M13), 7) ^ (M6))
```

Included in

Zvks, Zvksc, Zvksg, Zvksh

33.3.25. vsm4k.vi

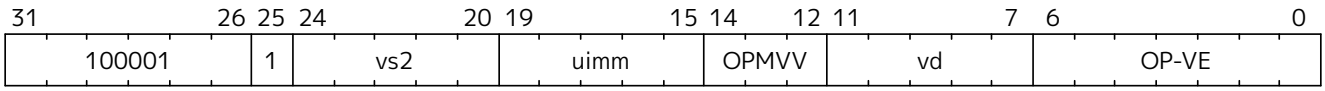
Synopsis

Vector SM4 KeyExpansion

Mnemonic

vsm4k.vi vd, vs2, uimm

Encoding



Reserved Encodings

- `SEW` is any value other than 32

Arguments

Register	Direction	EGW	EGS	EEW	Definition
uimm	input	-	-	-	Round group (rnd)
Vs2	input	128	4	32	Current 4 round keys rK[0:3]
Vd	output	128	4	32	Next 4 round keys rK'[0:3]

Description

Four rounds of the SM4 Key Expansion are performed.

Four round keys are read in as a 4-element group from `vs2`. Each of the next four round keys are generated by iteratively XORing the last three round keys with a constant that is indexed by the Round Group Number, performing a byte-wise substitution, and then performing XORs between rotated versions of this value and the corresponding current round key.

The Round group number (`rnd`) comes from `uimm[2:0]`; the bits in `uimm[4:3]` are ignored. Round group numbers range from 0 to 7 and indicate which group of four round keys are being generated. Round Keys range from 0-31. For example, if `rnd`=1, then round keys 4, 5, 6, and 7 are being generated.

Software needs to generate the initial round keys. This is done by XORing the 128-bit encryption key with the system parameters: FK[0:3]

Table 69. System Parameters

FK	constant
0	A3B1BAC6
1	56AA3350
2	677D9197
3	B27022DC

Implementation Hint

The round constants (CK) can be generated on the fly fairly cheaply. If the bytes of the constants are assigned an incrementing index from 0 to 127, the value of each byte is equal to its index multiplied by 7 modulo 256. Since the results are all limited to 8 bits, the modulo operation occurs for free:

```
B[n] = n + 2n + 4n;
     = 8n + ~n + 1;
```

Operation

```
function clause execute (vsm4k(uimm, vs2)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

  let B : bits(32) = 0;
  let S : bits(32) = 0;
  let rk4 : bits(32) = 0;
  let rk5 : bits(32) = 0;
  let rk6 : bits(32) = 0;
  let rk7 : bits(32) = 0;
  let rnd : bits(3) = uimm[2:0]; // Lower 3 bits

  foreach (i from eg_start to eg_len-1) {
    let (rk3 @ rk2 @ rk1 @ rk0) : bits(128) = get_velem(vs2, 128, i);

    B = rk1 ^ rk2 ^ rk3 ^ ck(4 * rnd);
    S = sm4_subword(B);
    rk4 = ROUND_KEY(rk0, S);

    B = rk2 ^ rk3 ^ rk4 ^ ck(4 * rnd + 1);
    S = sm4_subword(B);
    rk5 = ROUND_KEY(rk1, S);

    B = rk3 ^ rk4 ^ rk5 ^ ck(4 * rnd + 2);
    S = sm4_subword(B);
    rk6 = ROUND_KEY(rk2, S);

    B = rk4 ^ rk5 ^ rk6 ^ ck(4 * rnd + 3);
    S = sm4_subword(B);
    rk7 = ROUND_KEY(rk3, S);

    // Update the destination register.
   set_velem(vd, EGW=128, i, (rk7 @ rk6 @ rk5 @ rk4));
  }
  RETIRE_SUCCESS
  }
}
```

```
val round_key : bits(32) -> bits(32)
function ROUND_KEY(X, S) = ((X) ^ ((S) ^ ROL32((S), 13) ^ ROL32((S), 23)))

// SM4 Constant Key (CK)
let ck : list(bits(32)) = [|
 0x00070E15, 0x1C232A31, 0x383F464D, 0x545B6269,
 0x70777E85, 0x8C939AA1, 0xA8AFB6BD, 0xC4CBD2D9,
 0xE0E7EEF5, 0xFC030A11, 0x181F262D, 0x343B4249,
 0x50575E65, 0x6C737A81, 0x888F969D, 0xA4ABB2B9,
 0xC0C7CED5, 0xDCE3EAF1, 0xF8FF060D, 0x141B2229,
 0x30373E45, 0x4C535A61, 0x686F767D, 0x848B9299,
 0xA0A7AEB5, 0xBCC3CAD1, 0xD8DFE6ED, 0xF4FB0209,
 0x10171E25, 0x2C333A41, 0x484F565D, 0x646B7279
  |]
};
```

Included in

Zvks, Zvksc, Zvksed, Zvksg

33.3.26. vsm4r.[vv,vs]

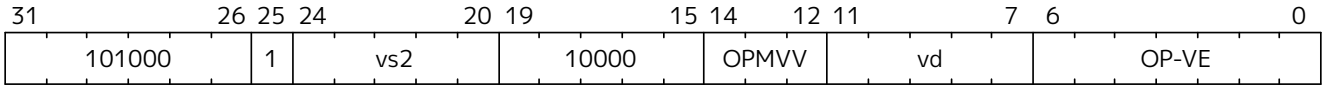
Synopsis

Vector SM4 Rounds

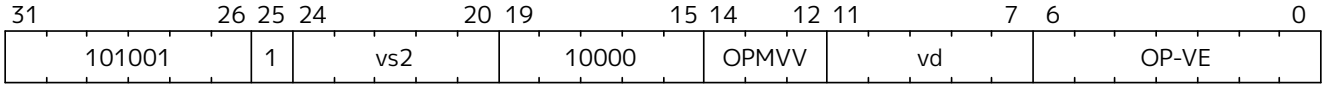
Mnemonic

vsm4r.vv vd, vs2
vsm4r.vs vd, vs2

Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Reserved Encodings

- `SEW` is any value other than 32
- Only for the `.vs` form: the `vd` register group overlaps the `vs2` register

Arguments

Register	Direction	EGW	EGS	EEW	Definition
Vd	input	128	4	32	Current state X[0:3]
Vs2	input	128	4	32	Round keys rk[0:3]
Vd	output	128	4	32	Next state X'[0:3]

Description

Four rounds of SM4 Encryption/Decryption are performed.

The four words of current state are read as a 4-element group from 'vd' and the round keys are read from either the corresponding 4-element group in `vs2` (vector-vector form) or the scalar element group in `vs2` (vector-scalar form). The next four words of state are generated by iteratively XORing the last three words of the state with the corresponding round key, performing a byte-wise substitution, and then performing XORs between rotated versions of this value and the corresponding current state.

In SM4, encryption and decryption are identical except that decryption consumes the round keys in the reverse order.

For the first four rounds of encryption, the current state is the plain text. For the first four rounds of decryption, the current state is the cipher text. For all subsequent rounds, the current state is the next state from the previous four rounds.

Operation

```
function clause execute (VSM4R(vd, vs2)) = {
  if(LMUL*VLEN < EGW)  then {
    handle_illegal();  // illegal-instruction exception
    RETIRE_FAIL
```

```
  } else {

  eg_len = (vl/EGS)
  eg_start = (vstart/EGS)

 let B  : bits(32) = 0;
 let S  : bits(32) = 0;
 let rk0 : bits(32) = 0;
 let rk1 : bits(32) = 0;
 let rk2 : bits(32) = 0;
 let rk3 : bits(32) = 0;
 let x0 : bits(32) = 0;
 let x1 : bits(32) = 0;
 let x2 : bits(32) = 0;
 let x3 : bits(32) = 0;
 let x4 : bits(32) = 0;
 let x5 : bits(32) = 0;
 let x6 : bits(32) = 0;
 let x7 : bits(32) = 0;

 let keyelem : bits(32) = 0;

  foreach (i from eg_start to eg_len-1) {
    keyelem = if suffix == "vv" then i else 0;
    {rk3 @ rk2 @ rk1 @ rk0} : bits(128) = get_velem(vs2, EGW=128, keyelem);
    {x3 @ x2 @ x1 @ x0} : bits(128) = get_velem(vd, EGW=128, i);

    B  = x1 ^ x2 ^ x3 ^ rk0;
    S = sm4_subword(B);
    x4 = sm4_round(x0, S);

    B = x2 ^ x3 ^ x4 ^ rk1;
    S = sm4_subword(B);
    x5= sm4_round(x1, S);

    B = x3 ^ x4 ^ x5 ^ rk2;
    S = sm4_subword(B);
    x6 = sm4_round(x2, S);

    B = x4 ^ x5 ^ x6 ^ rk3;
    S = sm4_subword(B);
    x7 = sm4_round(x3, S);

    set_velem(vd, EGW=128, i, (x7 @ x6 @ x5 @ x4));

  }
  RETIRE_SUCCESS
  }
}
```

```
val sm4_round : bits(32) -> bits(32)
function sm4_round(X, S) = \
  ((X) ^ ((S) ^ ROL32((S), 2) ^ ROL32((S), 10) ^ ROL32((S), 18) ^ ROL32((S),
24)))
```

Included in

Zvks, Zvksc, Zvksed, Zvksg

33.3.27. vwsll.[vv,vx,vi]

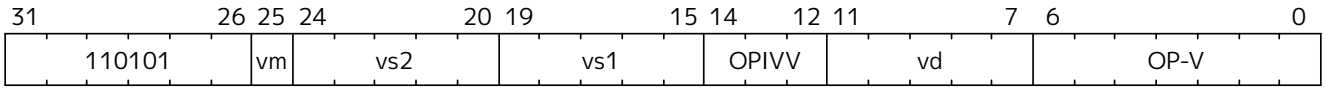
Synopsis

Vector widening shift left logical by vector/scalar/immediate.

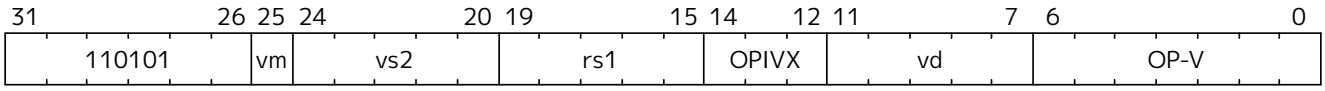
Mnemonic

vwsll.vv vd, vs2, vs1, vm
vwsll.vx vd, vs2, rs1, vm
vwsll.vi vd, vs2, uimm, vm

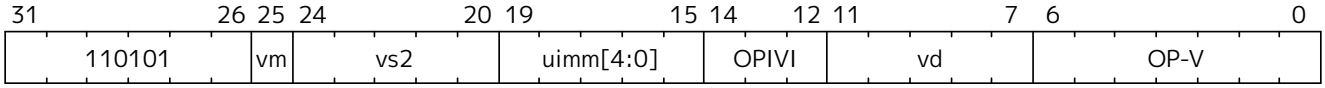
Encoding (Vector-Vector)



Encoding (Vector-Scalar)



Encoding (Vector-Immediate)



Vector-Vector Arguments

Register	Direction	Definition
Vs1	input	Shift amount
Vs2	input	Data
Vd	output	Shifted data

Vector-Scalar/Immediate Arguments

Register	Direction	EEW	Definition
Rs1/imm	input	SEW	Shift amount
Vs2	input	SEW	Data
Vd	output	2*SEW	Shifted data

Description

A widening logical shift left is performed on each element of `vs2`.

The elements in `vs2` are zero-extended to 2*`SEW` bits, then shifted left by the shift amount specified by either the corresponding elements of `vs1` (vector-vector), integer register `rs1` (vector-scalar), or an immediate value (vector-immediate). Only the low log2(2*`SEW`) bits of the shift-amount value are used, all other bits are ignored.

Operation

```
function clause execute (VWSLL_VV(vs2, vs1, vd)) = {
```

```
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (get_velem(vs1, i) & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VWSLL_VX(vs2, rs1, vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (X(rs1) & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}

function clause execute (VWSLL_VI(vs2, uimm[4:0], vd)) = {
  foreach (i from vstart to vl - 1) {
    set_velem(vd, EEW=2*SEW, i,
      get_velem(vs2, i) << (uimm[4:0] & ((2*SEW)-1))
    )
  }
  RETIRE_SUCCESS
}
```

Included in

Zvbb

33.4. Crypto Vector Cryptographic Instructions

OP-VE (0x77) Crypto Vector instructions except Zvbb and Zvbc

Integer					Integer				FP			
funct3					funct3				funct3			
OPIVV	V				OPMVV	V			OPFVV	V		
OPIVX		X			OPMVX		X		OPFVF		F	
OPIVI			I									

funct6					funct6				funct6			
100000					100000	V		vsm3me	100000			
100001					100001	V		vsm4k.vi	100001			
100010					100010	V		vaeskf1.vi	100010			
100011					100011				100011			
100100					100100				100100			
100101					100101				100101			
100110					100110				100110			
100111					100111				100111			
101000					101000	V		**VAES.vv**	101000			
101001					101001	V		**VAES.vs**	101001			
101010					101010	V		vaeskf2.vi	101010			
101011					101011	V		vsm3c.vi	101011			
101100					101100	V		vghsh	101100			
101101					101101	V		vsha2ms	101101			
101110					101110	V		vsha2ch	101110			
101111					101111	V		vsha2cl	101111			

Table 70. VAES.vv and VAES.vs encoding space

vs1	
00000	vaesdm
00001	vaesdf
00010	vaesem
00011	vaesef
00111	vaesz
10000	vsm4r
10001	vgmul

33.5. Vector Bitmanip and Carry-less Multiply Instructions

OP-V (0x57) **Zvbb**, **Zvkb**, and **Zvbc** Vector instructions **in bold**

Integer					Integer				FP				
funct3					funct3				funct3				
OPIVV	V				OPMVV	V			OPFVV	V			
OPIVX		X			OPMVX		X		OPFVF		F		
OPIVI			I										

funct6					funct6				funct6			
000000	V	X	I	vadd	000000	V		vredsum	000000	V	F	vfadd
000001	V	X		**vandn**	000001	V		vredand	000001	V		vfredusum
000010	V	X		vsub	000010	V		vredor	000010	V	F	vfsub
000011		X	I	vrsub	000011	V		vredxor	000011	V		vfredosum
000100	V	X		vminu	000100	V		vredminu	000100	V	F	vfmin
000101	V	X		vmin	000101	V		vredmin	000101	V		vfredmin
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F	vfmax
000111	V	X		vmax	000111	V		vredmax	000111	V		vfredmax
001000					001000	V	X	vaaddu	001000	V	F	vfsgnj
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F	vfsgnjn
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F	vfsgnjx
001011	V	X	I	vxor	001011	V	X	vasub	001011			
001100	V	X	I	vrgather	001100	V	X	**vclmul**	001100			
001101					001101	V	X	**vclmulh**	001101			
001110		X	I	vslideup	001110		X	vslide1up	001110		F	vfslide1up
001110	V			vrgatherei16								
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

funct6					funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXUNARY0	010000	V		VWFUNARY0
					010000		X	VRXUNARY0	010000		F	VRFUNARY0

funct6					funct6				funct6			
010001	V	X	I	vmadc	010001				010001			
010010	V	X		vsbc	010010	V		VXUNARY0	010010	V		VFUNARY0
010011	V	X		vmsbc	010011				010011	V		VFUNARY1
010100	V	X		**vror**	010100	V		VMUNARY0	010100			
010101	V	X		**vrol**	010101				010101			
01010x			I	**vror**								
010110					010110				010110			
010111	V	X	I	vmerge/vmv	010111	V		vcompress	010111		F	vfmerge/vfmv
011000	V	X	I	vmseq	011000	V		vmandn	011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		vmand	011001	V	F	vmfle
011010	V	X		vmsltu	011010	V		vmor	011010			
011011	V	X		vmslt	011011	V		vmxor	011011	V	F	vmflt
011100	V	X	I	vmsleu	011100	V		vmorn	011100	V	F	vmfne
011101	V	X	I	vmsle	011101	V		vmnand	011101		F	vmfgt
011110		X	I	vmsgtu	011110	V		vmnor	011110			
011111		X	I	vmsgt	011111	V		vmxnor	011111		F	vmfge

funct6					funct6				funct6			
100000	V	X	I	vsaddu	100000	V	X	vdivu	100000	V	F	vfdiv
100001	V	X	I	vsadd	100001	V	X	vdiv	100001		F	vfrdiv
100010	V	X		vssubu	100010	V	X	vremu	100010			
100011	V	X		vssub	100011	V	X	vrem	100011			
100100					100100	V	X	vmulhu	100100	V	F	vfmul
100101	V	X	I	vsll	100101	V	X	vmul	100101			
100110					100110	V	X	vmulhsu	100110			
100111	V	X		vsmul	100111	V	X	vmulh	100111		F	vfrsub
			I	vmv<nr>r								
101000	V	X	I	vsrl	101000				101000	V	F	vfmadd
101001	V	X	I	vsra	101001	V	X	vmadd	101001	V	F	vfnmadd
101010	V	X	I	vssrl	101010				101010	V	F	vfmsub
101011	V	X	I	vssra	101011	V	X	vnmsub	101011	V	F	vfnmsub
101100	V	X	I	vnsrl	101100				101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	vmacc	101101	V	F	vfnmacc
101110	V	X	I	vnclipu	101110				101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	vnmsac	101111	V	F	vfnmsac

funct6					funct6				funct6			
110000	V			vwredsumu	110000	V	X	vwaddu	110000	V	F	vfwadd
110001	V			vwredsum	110001	V	X	vwadd	110001	V		vfwredusum
110010					110010	V	X	vwsubu	110010	V	F	vfwsub

funct6					funct6				funct6			
110011					110011	V	X	vwsub	110011	V		vfwredosum
110100					110100	V	X	vwaddu.w	110100	V	F	vfwadd.w
110101	V	X	I	**vwsll**	110101	V	X	vwadd.w	110101			
110110					110110	V	X	vwsubu.w	110110	V	F	vfwsub.w
110111					110111	V	X	vwsub.w	110111			
111000					111000	V	X	vwmulu	111000	V	F	vfwmul
111001					111001				111001			
111010					111010	V	X	vwmulsu	111010			
111011					111011	V	X	vwmul	111011			
111100					111100	V	X	vwmaccu	111100	V	F	vfwmacc
111101					111101	V	X	vwmacc	111101	V	F	vfwnmacc
111110					111110		X	vwmaccus	111110	V	F	vfwmsac
111111					111111	V	X	vwmaccsu	111111	V	F	vfwnmsac

Table 71. VXUNARY0 encoding space

vs1	
00010	vzext.vf8
00011	vsext.vf8
00100	vzext.vf4
00101	vsext.vf4
00110	vzext.vf2
00111	vsext.vf2
01000	vbrev8
01001	vrev8
01010	vbrev
01100	vclz
01101	vctz
01110	vcpop

33.6. Supporting Sail Code

This section contains the supporting Sail code referenced by the instruction descriptions throughout the specification. The Sail Manual is recommended reading in order to best understand the supporting code.

```
/* Auxiliary function for performing GF multiplication */
val xt2 : bits(8) -> bits(8)
function xt2(x) = {
  (x << 1) ^ (if bit_to_bool(x[7]) then 0x1b else 0x00)
}

val xt3 : bits(8) -> bits(8)
function xt3(x) = x ^ xt2(x)

/* Multiply 8-bit field element by 4-bit value for AES MixCols step */
val gfmul : (bits(8), bits(4)) -> bits(8)
function gfmul( x, y) = {
  (if bit_to_bool(y[0]) then             x    else 0x00) ^
  (if bit_to_bool(y[1]) then xt2(        x)   else 0x00) ^
  (if bit_to_bool(y[2]) then xt2(xt2(    x))  else 0x00) ^
  (if bit_to_bool(y[3]) then xt2(xt2(xt2(x))) else 0x00)
}

/* 8-bit to 32-bit partial AES Mix Column - forwards */
val aes_mixcolumn_byte_fwd : bits(8) -> bits(32)
function aes_mixcolumn_byte_fwd(so) = {
  gfmul(so, 0x3) @ so @ so @ gfmul(so, 0x2)
}

/* 8-bit to 32-bit partial AES Mix Column - inverse*/
val aes_mixcolumn_byte_inv : bits(8) -> bits(32)
```

```
function aes_mixcolumn_byte_inv(so) = {
  gfmul(so, 0xb) @ gfmul(so, 0xd) @ gfmul(so, 0x9) @ gfmul(so, 0xe)
}

/* 32-bit to 32-bit AES forward MixColumn */
val aes_mixcolumn_fwd : bits(32) -> bits(32)
function aes_mixcolumn_fwd(x) = {
  let s0 : bits (8) = x[ 7.. 0];
  let s1 : bits (8) = x[15.. 8];
  let s2 : bits (8) = x[23..16];
  let s3 : bits (8) = x[31..24];
  let b0 : bits (8) = xt2(s0) ^ xt3(s1) ^    (s2) ^    (s3);
  let b1 : bits (8) =    (s0) ^ xt2(s1) ^ xt3(s2) ^    (s3);
  let b2 : bits (8) =    (s0) ^    (s1) ^ xt2(s2) ^ xt3(s3);
  let b3 : bits (8) = xt3(s0) ^    (s1) ^    (s2) ^ xt2(s3);
  b3 @ b2 @ b1 @ b0 /* Return value */
}

/* 32-bit to 32-bit AES inverse MixColumn */
val aes_mixcolumn_inv : bits(32) -> bits(32)
function aes_mixcolumn_inv(x) = {
  let s0 : bits (8) = x[ 7.. 0];
  let s1 : bits (8) = x[15.. 8];
  let s2 : bits (8) = x[23..16];
  let s3 : bits (8) = x[31..24];
  let b0 : bits (8) = gfmul(s0, 0xE) ^ gfmul(s1, 0xB) ^ gfmul(s2, 0xD) ^ gfmul
(s3, 0x9);
  let b1 : bits (8) = gfmul(s0, 0x9) ^ gfmul(s1, 0xE) ^ gfmul(s2, 0xB) ^ gfmul
(s3, 0xD);
  let b2 : bits (8) = gfmul(s0, 0xD) ^ gfmul(s1, 0x9) ^ gfmul(s2, 0xE) ^ gfmul
(s3, 0xB);
  let b3 : bits (8) = gfmul(s0, 0xB) ^ gfmul(s1, 0xD) ^ gfmul(s2, 0x9) ^ gfmul
(s3, 0xE);
  b3 @ b2 @ b1 @ b0 /* Return value */
}

val aes_decode_rcon : bits(4) -> bits(32)
function aes_decode_rcon(r) = {
  match r {
    0x0 => 0x00000001,
    0x1 => 0x00000002,
    0x2 => 0x00000004,
    0x3 => 0x00000008,
    0x4 => 0x00000010,
    0x5 => 0x00000020,
    0x6 => 0x00000040,
    0x7 => 0x00000080,
    0x8 => 0x0000001b,
    0x9 => 0x00000036,
    0xA => 0x00000000,
```

```
    0xB => 0x00000000,
    0xC => 0x00000000,
    0xD => 0x00000000,
    0xE => 0x00000000,
    0xF => 0x00000000
  }
}

/* SM4 SBox - only one sbox for forwards and inverse */
let sm4_sbox_table : list(bits(8)) = [|
0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7, 0x16, 0xB6, 0x14, 0xC2, 0x28,
0xFB, 0x2C, 0x05, 0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3, 0xAA, 0x44,
0x13, 0x26, 0x49, 0x86, 0x06, 0x99, 0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98,
0x7A, 0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62, 0xE4, 0xB3, 0x1C, 0xA9,
0xC9, 0x08, 0xE8, 0x95, 0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6, 0x47,
0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA, 0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85,
0x4F, 0xA8, 0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B, 0xF8, 0xEB, 0x0F,
0x4B, 0x70, 0x56, 0x9D, 0x35, 0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2,
0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87, 0xD4, 0x00, 0x46, 0x57, 0x9F,
0xD3, 0x27, 0x52, 0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E, 0xEA, 0xBF,
0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5, 0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15,
0xA1, 0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55, 0xAD, 0x93, 0x32, 0x30,
0xF5, 0x8C, 0xB1, 0xE3, 0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60, 0xC0,
0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F, 0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD,
0x8E, 0x2F, 0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51, 0x8D, 0x1B, 0xAF,
0x92, 0xBB, 0xDD, 0xBC, 0x7F, 0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD, 0x2D, 0x74, 0xD0, 0x12, 0xB8,
0xE5, 0xB4, 0xB0, 0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E, 0x65, 0xB9,
0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84, 0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D,
0x20, 0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48
|]

let aes_sbox_fwd_table : list(bits(8)) = [|
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe,
0xd7, 0xab, 0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4,
0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7,
0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, 0x04, 0xc7, 0x23, 0xc3,
0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, 0x09,
0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
0x2f, 0x84, 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe,
0x39, 0x4a, 0x4c, 0x58, 0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92,
0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, 0xcd, 0x0c,
0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb, 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2,
0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5,
0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, 0xba, 0x78, 0x25,
0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86,
```

```
0xc1, 0x1d, 0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e,
0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42,
0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
|]

let aes_sbox_inv_table : list(bits(8)) = [|
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81,
0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e,
0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23,
0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, 0x08, 0x2e, 0xa1, 0x66,
0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, 0x72,
0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65,
0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46,
0x57, 0xa7, 0x8d, 0x9d, 0x84, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca,
0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91,
0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6,
0x73, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f,
0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2,
0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, 0x1f, 0xdd, 0xa8,
0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93,
0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb,
0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6,
0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
|]

/* Lookup function - takes an index and a list, and retrieves the
 * x'th element of that list.
 */
val sbox_lookup : (bits(8), list(bits(8))) -> bits(8)
function sbox_lookup(x, table) = {
  match (x, table) {
    (0x00, t0::tn) => t0,
    (   y, t0::tn) => sbox_lookup(x - 0x01, tn)
  }
}

/* Easy function to perform a forward AES SBox operation on 1 byte. */
val aes_sbox_fwd : bits(8) -> bits(8)
function aes_sbox_fwd(x) = sbox_lookup(x, aes_sbox_fwd_table)

/* Easy function to perform an inverse AES SBox operation on 1 byte. */
val aes_sbox_inv : bits(8) -> bits(8)
function aes_sbox_inv(x) = sbox_lookup(x, aes_sbox_inv_table)

/* AES SubWord function used in the key expansion
 * - Applies the forward sbox to each byte in the input word.
 */
```

```
val aes_subword_fwd : bits(32) -> bits(32)
function aes_subword_fwd(x) = {
  aes_sbox_fwd(x[31..24]) @
  aes_sbox_fwd(x[23..16]) @
  aes_sbox_fwd(x[15.. 8]) @
  aes_sbox_fwd(x[ 7.. 0])
}

/* AES Inverse SubWord function.
 * - Applies the inverse sbox to each byte in the input word.
 */
val aes_subword_inv : bits(32) -> bits(32)
function aes_subword_inv(x) = {
  aes_sbox_inv(x[31..24]) @
  aes_sbox_inv(x[23..16]) @
  aes_sbox_inv(x[15.. 8]) @
  aes_sbox_inv(x[ 7.. 0])
}

/* Easy function to perform an SM4 SBox operation on 1 byte. */
val sm4_sbox : bits(8) -> bits(8)
function sm4_sbox(x) = sbox_lookup(x, sm4_sbox_table)

val aes_get_column : (bits(128), nat) -> bits(32)
function aes_get_column(state,c) = (state >> (to_bits(7, 32 * c)))[31..0]

/* 64-bit to 64-bit function which applies the AES forward sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_fwd_sbox_to_each_byte : bits(64) -> bits(64)
function aes_apply_fwd_sbox_to_each_byte(x) = {
  aes_sbox_fwd(x[63..56]) @
  aes_sbox_fwd(x[55..48]) @
  aes_sbox_fwd(x[47..40]) @
  aes_sbox_fwd(x[39..32]) @
  aes_sbox_fwd(x[31..24]) @
  aes_sbox_fwd(x[23..16]) @
  aes_sbox_fwd(x[15.. 8]) @
  aes_sbox_fwd(x[ 7.. 0])
}

/* 64-bit to 64-bit function which applies the AES inverse sbox to each byte
 * in a 64-bit word.
 */
val aes_apply_inv_sbox_to_each_byte : bits(64) -> bits(64)
function aes_apply_inv_sbox_to_each_byte(x) = {
  aes_sbox_inv(x[63..56]) @
  aes_sbox_inv(x[55..48]) @
  aes_sbox_inv(x[47..40]) @
  aes_sbox_inv(x[39..32]) @
```

```
  aes_sbox_inv(x[31..24]) @
  aes_sbox_inv(x[23..16]) @
  aes_sbox_inv(x[15.. 8]) @
  aes_sbox_inv(x[ 7.. 0])
}

/*
 * AES full-round transformation functions.
 */

val getbyte : (bits(64), int) -> bits(8)
function getbyte(x, i) = (x >> to_bits(6, i * 8))[7..0]

val aes_rv64_shiftrows_fwd : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_fwd(rs2, rs1) = {
  getbyte(rs1, 3) @
  getbyte(rs2, 6) @
  getbyte(rs2, 1) @
  getbyte(rs1, 4) @
  getbyte(rs2, 7) @
  getbyte(rs2, 2) @
  getbyte(rs1, 5) @
  getbyte(rs1, 0)
}

val aes_rv64_shiftrows_inv : (bits(64), bits(64)) -> bits(64)
function aes_rv64_shiftrows_inv(rs2, rs1) = {
  getbyte(rs2, 3) @
  getbyte(rs2, 6) @
  getbyte(rs1, 1) @
  getbyte(rs1, 4) @
  getbyte(rs1, 7) @
  getbyte(rs2, 2) @
  getbyte(rs2, 5) @
  getbyte(rs1, 0)
}

/* 128-bit to 128-bit implementation of the forward AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits  7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_fwd : bits(128) -> bits(128)
function aes_shift_rows_fwd(x) = {
  let ic3 : bits(32) = aes_get_column(x, 3);
  let ic2 : bits(32) = aes_get_column(x, 2);
  let ic1 : bits(32) = aes_get_column(x, 1);
  let ic0 : bits(32) = aes_get_column(x, 0);
  let oc0 : bits(32) = ic3[31..24] @ ic2[23..16] @ ic1[15.. 8] @ ic0[ 7.. 0];
  let oc1 : bits(32) = ic0[31..24] @ ic3[23..16] @ ic2[15.. 8] @ ic1[ 7.. 0];
  let oc2 : bits(32) = ic1[31..24] @ ic0[23..16] @ ic3[15.. 8] @ ic2[ 7.. 0];
```

```
  let oc3 : bits(32) = ic2[31..24] @ ic1[23..16] @ ic0[15.. 8] @ ic3[ 7.. 0];
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* 128-bit to 128-bit implementation of the inverse AES ShiftRows transform.
 * Byte 0 of state is input column 0, bits  7..0.
 * Byte 5 of state is input column 1, bits 15..8.
 */
val aes_shift_rows_inv : bits(128) -> bits(128)
function aes_shift_rows_inv(x) = {
  let ic3 : bits(32) = aes_get_column(x, 3); /* In column 3 */
  let ic2 : bits(32) = aes_get_column(x, 2);
  let ic1 : bits(32) = aes_get_column(x, 1);
  let ic0 : bits(32) = aes_get_column(x, 0);
  let oc0 : bits(32) = ic1[31..24] @ ic2[23..16] @ ic3[15.. 8] @ ic0[ 7.. 0];
  let oc1 : bits(32) = ic2[31..24] @ ic3[23..16] @ ic0[15.. 8] @ ic1[ 7.. 0];
  let oc2 : bits(32) = ic3[31..24] @ ic0[23..16] @ ic1[15.. 8] @ ic2[ 7.. 0];
  let oc3 : bits(32) = ic0[31..24] @ ic1[23..16] @ ic2[15.. 8] @ ic3[ 7.. 0];
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the forward sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_fwd : bits(128) -> bits(128)
function aes_subbytes_fwd(x) = {
  let oc0 : bits(32) = aes_subword_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_fwd(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse sub-bytes step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_subbytes_inv : bits(128) -> bits(128)
function aes_subbytes_inv(x) = {
  let oc0 : bits(32) = aes_subword_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_subword_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_subword_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_subword_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the forward MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_fwd : bits(128) -> bits(128)
function aes_mixcolumns_fwd(x) = {
```

```
  let oc0 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_fwd(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Applies the inverse MixColumns step of AES to a 128-bit vector
 * representation of its state.
 */
val aes_mixcolumns_inv : bits(128) -> bits(128)
function aes_mixcolumns_inv(x) = {
  let oc0 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 0));
  let oc1 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 1));
  let oc2 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 2));
  let oc3 : bits(32) = aes_mixcolumn_inv(aes_get_column(x, 3));
  (oc3 @ oc2 @ oc1 @ oc0) /* Return value */
}

/* Performs the word rotation for AES key schedule
*/

val aes_rotword : bits(32) -> bits(32)
function aes_rotword(x) = {
  let a0 : bits (8) = x[ 7.. 0];
  let a1 : bits (8) = x[15.. 8];
  let a2 : bits (8) = x[23..16];
  let a3 : bits (8) = x[31..24];
  (a0 @ a3 @ a2 @ a1) /* Return Value */
}

val brev : bits(SEW) -> bits(SEW)
function brev(x) = {
  let output : bits(SEW) = 0;
  foreach (i from 0 to SEW-8 by 8)
    output[i+7..i] = reverse_bits_in_byte(input[i+7..i]);
  output /* Return Value */
}

val reverse_bits_in_byte : bits(8) -> bits(8)
function reverse_bits_in_byte(x) = {
  let output : bits(8) = 0;
  foreach (i from 0 to 7)
    output[i] = x[7-i]);
  output /* Return Value */
}

val rev8 : bits(SEW) -> bits(SEW)
function rev8(x) = {     // endian swap
  let output : bits(SEW) = 0;
```

```
    let j = SEW - 1;
    foreach (k from 0 to (SEW - 8) by 8) {
      output[k..(k + 7)] = x[(j - 7)..j];
      j = j - 8;
  output /* Return Value */
  }
  RETIRE_SUCCESS


val rol32 : bits(32) -> bits(32)
function ROL32(x,n) = (X << N) | (X >> (32 - N))

val sm4_subword : bits(32) -> bits(32)
function sm4_subword(x) = {
  sm4_sbox(x[31..24]) @
  sm4_sbox(x[23..16]) @
  sm4_sbox(x[15.. 8]) @
  sm4_sbox(x[ 7.. 0])
}
```

Chapter 34. Control-flow Integrity (CFI)

Control-flow Integrity (CFI) capabilities help defend against Return-Oriented Programming (ROP) and Call/Jump-Oriented Programming (COP/JOP) style control-flow subversion attacks. These attack methodologies use code sequences in authorized modules, with at least one instruction in the sequence being a control transfer instruction that depends on attacker-controlled data either in the return stack or in memory used to obtain the target address for a call or jump. Attackers stitch these sequences together by diverting the control flow instructions (e.g., `JALR`, `C.JR`, `C.JALR`), from their original target address to a new target via modification in the return stack or in the memory used to obtain the jump/call target address.

RV32/RV64 provides two types of control transfer instructions - unconditional jumps and conditional branches. Conditional branches encode an offset in the immediate field of the instruction and are thus direct branches that are not susceptible to control-flow subversion. Unconditional direct jumps using `JAL` transfer control to a target that is in a +/- 1 MiB range from the current `pc`. Unconditional indirect jumps using the `JALR` obtain their branch target by adding the sign extended 12-bit immediate encoded in the instruction to the `rs1` register.

The RV32I/RV64I does not have a dedicated instruction for calling a procedure or returning from a procedure. A `JAL` or `JALR` may be used to perform a procedure call and `JALR` to return from a procedure. The RISC-V ABI however defines the convention that a `JAL`/`JALR` where `rd` (i.e. the link register) is `x1` or `x5` is a procedure call, and a `JALR` where `rs1` is the conventional link register (i.e. `x1` or `x5`) is a return from procedure. The architecture allows for using these hints and conventions to support return address prediction (See Table 3).

The RVC standard extension for compressed instructions provides unconditional jump and conditional branch instructions. The `C.J` and `C.JAL` instructions encode an offset in the immediate field of the instruction and thus are not susceptible to control-flow subversion. The `C.JR` and `C.JALR` RVC instructions perform an unconditional control transfer to the address in register `rs1`. The `C.JALR` additionally writes the address of the instruction following the jump (`pc+2`) to the link register `x1` and is a procedure call. The `C.JR` is a return from procedure if `rs1` is a conventional link register (i.e. `x1` or `x5`); else it is an indirect jump.

The term *call* is used to refer to a `JAL` or `JALR` instruction with a link register as destination, i.e., *rd*≠`x0`. Conventionally, the link register is `x1` or `x5`. A *call* using `JAL` or `C.JAL` is termed a direct call. A `C.JALR` expands to `JALR x1, 0(rs1)` and is a *call*. A *call* using `JALR` or `C.JALR` is termed an *indirect-call*.

The term *return* is used to refer to a `JALR` instruction with *rd*=`x0` and with *rs1*=`x1` or *rs1*=`x5`. A `C.JR` instruction expands to `JALR x0, 0(rs1)` and is a *return* if *rs1*=`x1` or *rs1*=`x5`.

The term *indirect-jump* is used to refer to a `JALR` instruction with *rd*=`x0` and where the *rs1* is not `x1` or `x5` (i.e., not a return). A `C.JR` instruction where *rs1* is not `x1` or `x5` (i.e., not a return) is an *indirect-jump*.

The Zicfiss and Zicfilp extensions build on these conventions and hints and provide backward-edge and forward-edge control flow integrity respectively.

The Unprivileged ISA for Zicfilp extension is specified in Section 34.1 and for the Unprivileged ISA for Zicfiss extension is specified in Section 34.2. The Privileged ISA for these extensions is specified in the Privileged ISA specification.

34.1. Landing Pad (Zicfilp)

To enforce forward-edge control-flow integrity, the Zicfilp extension introduces a landing pad (`LPAD`) instruction. The `LPAD` instruction must be placed at the program locations that are valid targets of indirect jumps or calls. The `LPAD` instruction (See Section 34.1.2) is encoded using the `AUIPC` major opcode with *rd* =`x0`.

Compilers emit a landing pad instruction as the first instruction of an address-taken function, as well as at any indirect jump targets. A landing pad instruction is not required in functions that are only reached using a direct call or direct jump.

The landing pad is designed to provide integrity to control transfers performed using indirect calls and jumps, and this is referred to as forward-edge protection. When the Zicfilp is active, the hart tracks an expected landing pad (`ELP`) state that is updated by an *indirect_call* or *indirect_jump* to require a landing pad instruction at the target of the branch. If the instruction at the target is not a landing pad, then a software-check exception is raised.

A landing pad may be optionally associated with a 20-bit label. With labeling enabled, the number of landing pads that can be reached from an indirect call or jump sites can be defined using programming language-based policies. Labeling of the landing pads enables software to achieve greater precision in pairing up indirect call/jump sites with valid targets. When labeling of landing pads is used, indirect call or indirect jump site can specify the expected label of the landing pad and thereby constrain the set of landing pads that may be reached from each indirect call or indirect jump site in the program.

In the simplest form, a program can be built with a single label value to implement a coarse-grained version of forward-edge control-flow integrity. By constraining gadgets to be preceded by a landing pad instruction that marks the start of indirect callable functions, the program can significantly reduce the available gadget space. A second form of label generation may generate a signature, such as a MAC, using the prototype of the function. Programs that use this approach would further constrain the gadgets accessible from a call site to only indirectly callable functions that match the prototype of the called functions. Another approach to label generation involves analyzing the control-flow-graph (CFG) of the program, which can lead to even more stringent constraints on the set of reachable gadgets. Such programs may further use multiple labels per function, which means that if a function is called from two or more call sites, the functions can be labeled as being reachable from each of the call sites. For instance, consider two call sites A and B, where A calls the functions X and Y, and B calls the functions Y and Z. In a single label scheme, functions X, Y, and Z would need to be assigned the same label so that both call sites A and B can invoke the common function Y. This scheme would allow call site A to also call function Z and call site B to also call function X. However, if function Y was assigned two labels - one corresponding to call site A and the other to call site B, then Y can be invoked by both call sites, but X can only be invoked by call site A and Z can only be invoked by call site B. To support multiple labels, the compiler could generate a call-site-specific entry point for shared functions, with each entry point having its own landing pad instruction followed by a direct branch to the start of the function. This would allow the function to be labeled with multiple labels, each corresponding to a specific call site. A portion of the label space may be dedicated to labeled landing pads that are only valid targets of an indirect jump (and not an indirect call).

The `LPAD` instruction uses the code points defined as HINTs for the `AUIPC` opcode. When Zicfilp is not active at a privilege level or when the extension is not implemented, the landing pad instruction executes as a no-op. A program that is built with `LPAD` instructions can thus continue to operate correctly, but without forward-edge control-flow integrity, on processors that do not support the Zicfilp extension or if the Zicfilp extension is not active.

Compilers and linkers should provide an attribute flag to indicate if the program has been compiled with the Zicfilp extension and use that to determine if the Zicfilp extension should be activated. The dynamic loader should activate the use of Zicfilp extension for an application only if all executables (the application and the dependent dynamically linked libraries) used by that application use the Zicfilp extension.

When Zicfilp extension is not active or not implemented, the hart does not require landing pad instructions at the targets of indirect calls/jumps, and the landing instructions revert to being no-ops. This allows a program compiled with landing pad instructions to operate correctly but without forward-edge control-flow integrity.

The Zicfilp extensions may be activated for use individually and independently for each privilege mode.

The Zicfilp extension depends on the Zicsr extension.

34.1.1. Landing Pad Enforcement

To enforce that the target of an indirect call or indirect jump must be a valid landing pad instruction, the hart maintains an expected landing pad (`ELP`) state to determine if a landing pad instruction is required at the target of an indirect call or an indirect jump. The `ELP` state can be one of:

- 0 - `NO_LP_EXPECTED`
- 1 - `LP_EXPECTED`

The `ELP` state is initialized to `NO_LP_EXPECTED` by the hart upon reset.

The Zicfilp extension, when enabled, determines if an indirect call or an indirect jump must land on a landing pad, as specified in Listing 4. If `is_lp_expected` is 1, then the hart updates the `ELP` to `LP_EXPECTED`.

```
is_lp_expected = ( (JALR || C.JR || C.JALR) &&
                   (rs1 != x1) && (rs1 != x5) && (rs1 != x7) ) ? 1 : 0;
```

Listing 4. Landing pad expected determination

An indirect branch using `JALR`, `C.JALR`, or `C.JR` with `rs1` as `x7` is termed a software guarded branch. Such branches do not need to land on a `LPAD` instruction and thus do not set `ELP` to `LP_EXPECTED`.

When the register source is a link register and the register destination is `x0`, then it's a return from a procedure and does not require a landing pad at the target.

When the register source and register destination are both link registers, then it is a semantically-direct-call. For example, the `call offset` pseudoinstruction may expand to a two instruction sequence composed of a `lui ra, imm20` or a `auipc ra, imm20` instruction followed by a `jalr ra, imm12(ra)` instruction where `ra` is the link register (either `x1` or `x5`). Since the address of the procedure was not explicitly taken and the computed address is not obtained from mutable memory, such semantically-direct calls do not require a landing pad to be placed at the target. Compilers and JITers must use the semantically-direct calls only if the `rs1` was computed as a PC-relative or an absolute offset to the symbol.

The `tail offset` pseudoinstruction used to tail call a far-away procedure may also be expanded to a two instruction sequence composed of a `lui x7, imm20` or `auipc x7, imm20` followed by a `jalr x0, x7`. Since the address of the procedure was not explicitly taken and the computed address is not obtained from mutable memory, such semantically-direct tail-calls do not require a landing pad to be placed at the target.

Software guarded branches may also be used by compilers to generate code for constructs like switch-cases. When using the software guarded branches, the compiler is required to ensure it has full control on the possible jump targets (e.g., by obtaining the targets from a read-only table in memory and performing bounds checking on the index into the table, etc.).

The landing pad may be labeled. Zicfilp extension designates the register `x7` for use as the landing pad label register. To support labeled landing pads, the indirect call/jump sites establish an expected landing pad label (e.g., using the `LUI` instruction) in the bits 31:12 of the `x7` register. The `LPAD` instruction is encoded with a 20-bit immediate value called the landing-pad-label (`LPL`) that is matched to the expected landing pad label. When `LPL` is encoded as zero, the `LPAD` instruction does not perform the label check and in programs built with this single label mode of operation the indirect call/jump sites do not need to establish an

expected landing pad label value in `x7`.

When `ELP` is set to `LP_EXPECTED`, if the next instruction in the instruction stream is not 4-byte aligned, or is not `LPAD`, or if the landing pad label encoded in `LPAD` is not zero and does not match the expected landing pad label in bits 31:12 of the `x7` register, then a software-check exception (cause=18) with `xtval` set to "landing pad fault (code=2)" is raised else the `ELP` is updated to `NO_LP_EXPECTED`.

> *The tracking of* `ELP` *and the requirement for a landing pad instruction at the target of indirect call and jump enables a processor implementation to significantly reduce or to prevent speculation to non-landing-pad instructions. Constraining speculation using this technique, greatly reduces the gadget space and increases the difficulty of using techniques such as branch-target-injection, also known as Spectre variant 2, which use speculative execution to leak data through side channels.*

> *The* `LPAD` *requires a 4-byte alignment to address the concatenation of two instructions* `A` *and* `B` *accidentally forming an unintended landing pad in the program. For example, consider a 32-bit instruction where the bytes 3 and 2 have a pattern of* `?017h` *(for example, the immediate fields of a* `LUI`, `AUIPC`, *or a* `JAL` *instruction), followed by a 16-bit or a 32-bit instruction. When patterns that can accidentally form a valid landing pad are detected, the assembler or linker can force instruction* `A` *to be aligned to a 4-byte boundary to force the unintended* `LPAD` *pattern to become misaligned, and thus not a valid landing pad, or may use an alternate register allocation to prevent the accidental landing pad.*

34.1.2. Landing Pad Instruction

When Zicfilp is enabled, `LPAD` is the only instruction allowed to execute when the `ELP` state is `LP_EXPECTED`. If Zicfilp is not enabled then the instruction is a no-op. If Zicfilp is enabled, the `LPAD` instruction causes a software-check exception with `xtval` set to "landing pad fault (code=2)" if any of the following conditions are true:

- The `pc` is not 4-byte aligned and `ELP` is `LP_EXPECTED`.
- The `ELP` is `LP_EXPECTED` and the `LPL` is not zero and the `LPL` does not match the expected landing pad label in bits 31:12 of the `x7` register.

If a software-check exception is not caused then the `ELP` is updated to `NO_LP_EXPECTED`.

The operation of the `LPAD` instruction is as follows:

```
if (xLPE == 1 && ELP == LP_EXPECTED)
    // If PC not 4-byte aligned then software-check exception
    if pc[1:0] != 0
        raise software-check exception
    // If landing pad label not matched -> software-check exception
    else if (inst.LPL != x7[31:12] && inst.LPL != 0)
        raise software-check exception
    else
        ELP = NO_LP_EXPECTED
else
    no-op
endif
```

Listing 5. `LPAD` *operation*

<<<

34.2. Shadow Stack (Zicfiss)

The Zicfiss extension introduces a shadow stack to enforce backward-edge control-flow integrity. A shadow stack is a second stack used to store a shadow copy of the return address in the link register if it needs to be spilled.

The shadow stack is designed to provide integrity to control transfers performed using a *return*, where the return may be from a procedure invoked using an indirect call or a direct call, and this is referred to as backward-edge protection.

A program using backward-edge control-flow integrity has two stacks: a regular stack and a shadow stack. The shadow stack is used to spill the link register, if required, by non-leaf functions. An additional register, shadow-stack-pointer (`ssp`), is introduced in the architecture to hold the address of the top of the active shadow stack.

The shadow stack, similar to the regular stack, grows downwards, from higher addresses to lower addresses. Each entry on the shadow stack is `XLEN` wide and holds the link register value. The `ssp` points to the top of

the shadow stack, which is the address of the last element stored on the shadow stack.

The shadow stack is architecturally protected from inadvertent corruptions and modifications, as detailed in the Privileged specification.

The Zicfiss extension provides instructions to store and load the link register to/from the shadow stack and to check the integrity of the return address. The extension provides instructions to support common stack maintenance operations such as stack unwinding and stack switching.

When Zicfiss is enabled, each function that needs to spill the link register, typically non-leaf functions, store the link register value to the regular stack and a shadow copy of the link register value to the shadow stack when the function is entered (the prologue). When such a function returns (the epilogue), the function loads the link register from the regular stack and the shadow copy of the link register from the shadow stack. Then, the link register value from the regular stack and the shadow link register value from the shadow stack are compared. A mismatch of the two values is indicative of a subversion of the return address control variable and causes a software-check exception.

The Zicfiss instructions, except `SSAMOSWAP.W/D`, are encoded using a subset of May-Be-Operation instructions defined by the Zimop and Zcmop extensions. This subset of instructions revert to their Zimop/Zcmop defined behavior when the Zicfiss extension is not implemented or if the extension has not been activated. A program that is built with Zicfiss instructions can thus continue to operate correctly, but without backward-edge control-flow integrity, on processors that do not support the Zicfiss extension or if the Zicfiss extension is not active. The Zicfiss extension may be activated for use individually and independently for each privilege mode.

Compilers should flag each object file (for example, using flags in the ELF attributes) to indicate if the object file has been compiled with the Zicfiss instructions. The linker should flag (for example, using flags in the ELF attributes) the binary/executable generated by linking objects as being compiled with the Zicfiss instructions only if all the object files that are linked have the same Zicfiss attributes.

The dynamic loader should activate the use of Zicfiss extension for an application only if all executables (the application and the dependent dynamically-linked libraries) used by that application use the Zicfiss extension.

An application that has the Zicfiss extension active may request the dynamic loader at runtime to load a new dynamic shared object (using dlopen() for example). If the requested object does not have the Zicfiss attribute then the dynamic loader, based on its policy (e.g., established by the operating system or the administrator) configuration, could either deny the request or deactivate the Zicfiss extension for the application. It is strongly recommended that the policy enforces a strict security posture and denies the request.

The Zicfiss extension depends on the Zicsr, Zimop and Zaamo extensions. Furthermore, if the Zcmop extension is implemented, the Zicfiss extension also provides the `C.SSPUSH` and `C.SSPOPCHK` instructions. Moreover, use of Zicfiss in U-mode requires S-mode to be implemented. Use of Zicfiss in M-mode is not supported.

34.2.1. Zicfiss Instructions Summary

The Zicfiss extension introduces the following instructions:

- Push to the shadow stack (See Section 34.2.4)
  - `SSPUSH x1` and `SSPUSH x5` - encoded using `MOP.RR.7`
  - `C.SSPUSH x1` - encoded using `C.MOP.1`
- Pop from the shadow stack (See Section 34.2.5)
  - `SSPOPCHK x1` and `SSPOPCHK x5` - encoded using `MOP.R.28`
  - `C.SSPOPCHK x5` - encoded using `C.MOP.5`
- Read the value of `ssp` into a register (See Section 34.2.6)
  - `SSRDP` - encoded using `MOP.R.28`
- Perform an atomic swap from a shadow stack location (See Section 34.2.7)
  - `SSAMOSWAP.W` and `SSAMOSWAP.D`

Zicfiss does not use all encodings of `MOP.RR.7` or `MOP.R.28`. When a `MOP.RR.7` or `MOP.R.28` encoding is not used by the Zicfiss extension, the corresponding instruction adheres to its Zimop-defined behavior, unless redefined by another extension.

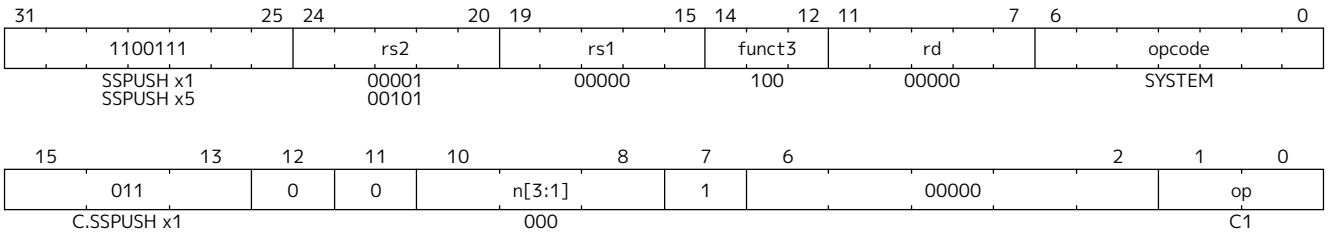
34.2.2. Shadow Stack Pointer (`ssp`)

The `ssp` CSR is an unprivileged read-write (URW) CSR that reads and writes `XLEN` low order bits of the shadow stack pointer (`ssp`). The CSR address is 0x011. There is no high CSR defined as the `ssp` is always as wide as the `XLEN` of the current privilege mode. The bits 1:0 of `ssp` are read-only zero. If the UXLEN or SXLEN may never be 32, then the bit 2 is also read-only zero.

34.2.3. Zicfiss Instructions

34.2.4. Push to the Shadow Stack

A shadow stack push operation is defined as decrement of the `ssp` by `XLEN/8` followed by a store of the value in the link register to memory at the new top of the shadow stack.



Only `x1` and `x5` registers are supported as `rs2` for `SSPUSH`. Zicfiss provides a 16-bit version of the `SSPUSH x1` instruction using the Zcmop defined `C.MOP.1` encoding. The `C.SSPUSH x1` expands to `SSPUSH x1`.

The `SSPUSH` instruction and its compressed form `C.SSPUSH` can be used to push a link register on the shadow stack. The `SSPUSH` and `C.SSPUSH` instructions perform a store identically to the existing store instructions, with the difference that the base is implicitly `ssp` and the width is implicitly `XLEN`.

The operation of the `SSPUSH` and `C.SSPUSH` instructions is as follows:

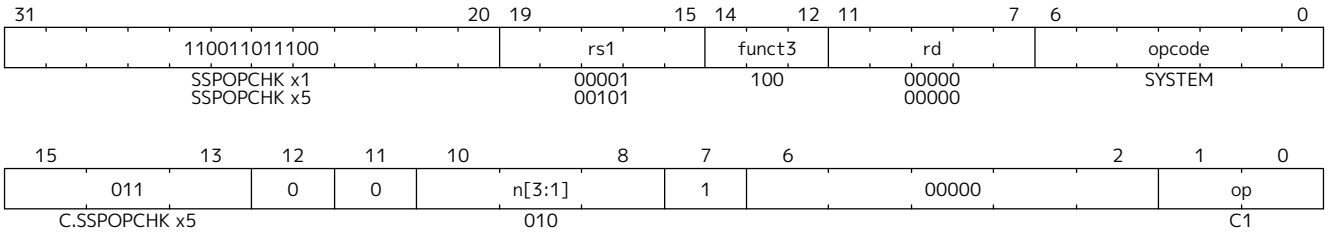
```
if (xSSE == 1)
    mem[ssp - (XLEN/8)] = X(src)  # Store src value to ssp - XLEN/8
    ssp = ssp - (XLEN/8)          # decrement ssp by XLEN/8
endif
```

Listing 6. `SSPUSH` *and* `C.SSPUSH` *operation*

The `ssp` is decremented by `SSPUSH` and `C.SSPUSH` only if the store to the shadow stack completes successfully.

34.2.5. Pop from the Shadow Stack

A shadow stack pop operation is defined as an `XLEN` wide read from the current top of the shadow stack followed by an increment of the `ssp` by `XLEN/8`.



Only `x1` and `x5` registers are supported as `rs1` for `SSPOPCHK`. Zicfiss provides a 16-bit version of the `SSPOPCHK x5` using the Zcmop defined `C.MOP.5` encoding. The `C.SSPOPCHK x5` expands to `SSPOPCHK x5`.

Programs with a shadow stack push the return address onto the regular stack as well as the shadow stack in the prologue of non-leaf functions. When returning from these non-leaf functions, such programs pop the link register from the regular stack and pop a shadow copy of the link register from the shadow stack. The two values are then compared. If the values do not match, it is indicative of a corruption of the return address variable on the regular stack.

The `SSPOPCHK` instruction, and its compressed form `C.SSPOPCHK`, can be used to pop the shadow return address value from the shadow stack and check that the value matches the contents of the link register, and if not cause a software-check exception with `xtval` set to "shadow stack fault (code=3)".

While any register may be used as link register, conventionally the `x1` or `x5` registers are used. The shadow stack instructions are designed to be most efficient when the `x1` and `x5` registers are used as the link register.

> *Return-address prediction stacks are a common feature of high-performance instruction-fetch units, but they require accurate detection of instructions used for procedure calls and returns to be effective. For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. The return-address stack (RAS) actions to pop and/or push onto the RAS are specified in Table 3.*
>
> *Using `x1` or `x5` as the link register allows a program to benefit from the return-address prediction stacks. Additionally, since the shadow stack instructions are designed around the use of `x1` or `x5` as the link register, using any other register as a link register would incur the cost of additional register movements.*
>
> *Compilers, when generating code with backward-edge CFI, must protect the link register, e.g., `x1` and/or `x5`, from arbitrary modification by not emitting unsafe code sequences.*

Storing the return address on both stacks preserves the call stack layout and the ABI, while also allowing for the detection of corruption of the return address on the regular stack. The prologue and epilogue of a non-leaf function that uses shadow stacks is as follows:

```
function_entry:
    addi sp,sp,-8  # push link register x1
    sd x1,(sp)     # on regular stack
    sspush x1      # push link register x1 on shadow stack
     :
    ld x1,(sp)     # pop link register x1 from regular stack
    addi sp,sp,8
    sspopchk x1    # fault if x1 not equal to shadow
                   # return address
    ret
```

This example illustrates the use of `x1` *register as the link register. Alternatively, the* `x5` *register may also be used as the link register.*

A leaf function, a function that does not itself make function calls, does not need to spill the link register. Consequently, the return value may be held in the link register itself for the duration of the leaf function's execution.

The `C.SSPOPCHK`, and `SSPOPCHK` instructions perform a load identically to the existing load instructions, with the difference that the base is implicitly `ssp` and the width is implicitly `XLEN`.

The operation of the `SSPOPCHK` and `C.SSPOPCHK` instructions is as follows:

```
if (xSSE == 1)
    temp = mem[ssp]            # Load temp from address in ssp and
    if temp != X(src)          # Compare temp to value in src and
                               # cause a software-check exception
                               # if they are not bitwise equal.
                               # Only x1 and x5 may be used as src
       raise software-check exception
    else
       ssp = ssp + (XLEN/8)    # increment ssp by XLEN/8.
    endif
endif
```

Listing 7. `SSPOPCHK` *and* `C.SSPOPCHK` *operation*

If the value loaded from the address in `ssp` does not match the value in `rs1`, a software-check exception (cause=18) is raised with `xtval` set to "shadow stack fault (code=3)". The software-check exception caused by `SSPOPCHK`/ `C.SSPOPCHK` is lower in priority than a load/store/AMO access-fault exception.

The `ssp` is incremented by `SSPOPCHK` and `C.SSPOPCHK` only if the load from the shadow stack completes successfully and no software-check exception is raised.

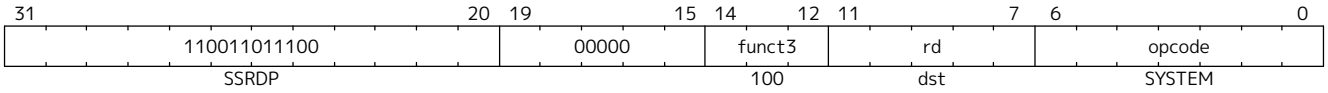
The use of the compressed instruction `C.SSPUSH x1` *to push on the shadow stack is most efficient when the ABI uses* `x1` *as the link register, as the link register may then be pushed without needing a register-to-register move in the function prologue. To use the compressed instruction* `C.SSPOPCHK x5`*, the function should pop the return address from regular stack into the alternate link register* `x5` *and use the* `C.SSPOPCHK x5` *to compare the return address to the shadow copy stored on the shadow stack. The function then uses* `C.JR x5` *to jump to the return address.*

```
    function_entry:
        c.addi sp,sp,-8  # push link register x1
        c.sd x1,(sp)     # on regular stack
        c.sspush x1      # push link register x1 on shadow stack
         :
        c.ld x5,(sp)     # pop link register x5 from regular
stack
        c.addi sp,sp,8
        c.sspopchk x5    # fault if x5 not equal to shadow return
address
        c.jr x5
```

Store-to-load forwarding is a common technique employed by high-performance processor implementations. Zicfiss implementations may prevent forwarding from a non-shadow-stack store to the `SSPOPCHK` *or the* `C.SSPOPCHK` *instructions. A non-shadow-stack store causes a fault if done to a page mapped as a shadow stack. However, such determination may be delayed till the PTE has been examined and thus may be used to transiently forward the data from such stores to* `SSPOPCHK` *or to* `C.SSPOPCHK`*.*

34.2.6. Read `ssp` into a Register

The `SSRDP` instruction is provided to move the contents of `ssp` to a destination register.



Encoding *rd* as `x0` is not supported for `SSRDP`.

The operation of the `SSRDP` instructions is as follows:

```
if (xSSE == 1)
    X(dst) = ssp
else
    X(dst) = 0
endif
```

Listing 8. `SSRDP` *operation*

The property of Zimop writing 0 to the `rd` *when the extension using Zimop is not implemented or not active may be used by to determine if Zicfiss extension is active. For example, functions that unwind shadow stacks may skip over the unwind actions by dynamically detecting if the Zicfiss extension is active.*

An example sequence such as the following may be used:

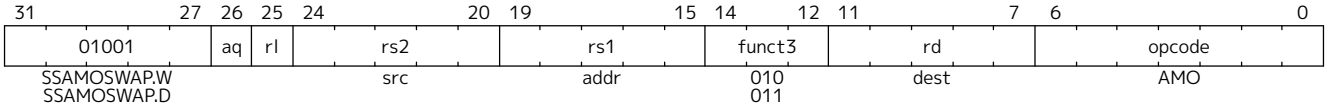
```
    ssrdp t0                        # mv ssp to t0
    beqz t0, zicfiss_not_active     # zero is not a valid shadow
stack
                                    # pointer by convention
    # Zicfiss is active
    :
    :
zicfiss_not_active:
```

To assist with the use of such code sequences, operating systems and runtimes must not locate shadow stacks at address 0.

A common operation performed on stacks is to unwind them to support constructs like `setjmp`/`longjmp`*, C++ exception handling, etc. A program that uses shadow stacks must unwind the shadow stack in addition to the stack used to store data. The unwind function must verify that it does not accidentally unwind past the bounds of the shadow stack. Shadow stacks are expected to be bounded on each end using guard pages. A guard page for a stack is a page that is not accessible by the process that owns the stack. To detect if the unwind occurs past the bounds of the shadow stack, the unwind may be done in maximal increments of 4 KiB, testing whether the* `ssp` *is still pointing to a shadow stack page or has unwound into the guard page. The following examples illustrate the use of shadow stack instructions to unwind a shadow stack. This example assumes that the* `setjmp` *function itself does not push on to the shadow stack (being a leaf function, it is not required to).*

```
setjmp() {
    :
    :
    // read and save the shadow stack pointer to jmp_buf
    asm("ssrdp %0" : "=r"(cur_ssp):);
    jmp_buf->saved_ssp = cur_ssp;
    :
    :
}
longjmp() {
    :
    // Read current shadow stack pointer and
    // compute number of call frames to unwind
    asm("ssrdp %0" : "=r"(cur_ssp):);
    // Skip the unwind if backward-edge CFI not active
    asm("beqz %0, back_cfi_not_active" : "=r"(cur_ssp):);
    // Unwind the frames in a loop
    while ( jmp_buf->saved_ssp > cur_ssp ) {
        // advance by a maximum of 4K at a time to avoid
        // unwinding past bounds of the shadow stack
        cur_ssp = ( (jmp_buf->saved_ssp - cur_ssp) >= 4096 ) ?
                  (cur_ssp + 4096) : jmp_buf->saved_ssp;
        asm("csrw ssp, %0" : :  "r" (cur_ssp));
        // Test if unwound past the shadow stack bounds
        asm("sspush x5");
        asm("sspopchk x5");
    }
back_cfi_not_active:
    :
}
```

34.2.7. Atomic Swap from a Shadow Stack Location



For RV32, `SSAMOSWAP.W` atomically loads a 32-bit data value from address of a shadow stack location in `rs1`, puts the loaded value into register `rd`, and stores the 32-bit value held in `rs2` to the original address in `rs1`. `SSAMOSWAP.D` (RV64 only) is similar to `SSAMOSWAP.W` but operates on 64-bit data values.

```
if privilege_mode != M && menvcfg.SSE == 0
    raise illegal-instruction exception
else if S-mode not implemented
    raise illegal-instruction exception
else if privilege_mode == U && senvcfg.SSE == 0
    raise illegal-instruction exception
else if privilege_mode == VS && henvcfg.SSE == 0
    raise virtual-instruction  exception
else if privilege_mode == VU && senvcfg.SSE == 0
    raise virtual-instruction  exception
else
    X(rd) = mem[X(rs1)]
    mem[X(rs1)] = X(rs2)
endif
```

Listing 9. `SSAMOSWAP.W` *for RV32 and* `SSAMOSWAP.D` *(RV64 only) operation*

For RV64, `SSAMOSWAP.W` atomically loads a 32-bit data value from address of a shadow stack location in `rs1`, sign-extends the loaded value and puts it in `rd`, and stores the lower 32 bits of the value held in `rs2` to the original address in `rs1`.

```
if privilege_mode != M && menvcfg.SSE == 0
    raise illegal-instruction exception
else if S-mode not implemented
    raise illegal-instruction exception
else if privilege_mode == U && senvcfg.SSE == 0
    raise illegal-instruction exception
else if privilege_mode == VS && henvcfg.SSE == 0
    raise virtual-instruction  exception
else if privilege_mode == VU && senvcfg.SSE == 0
    raise virtual-instruction  exception
else
    temp[31:0] = mem[X(rs1)]
    X(rd) = SignExtend(temp[31:0])
    mem[X(rs1)] = X(rs2)[31:0]
endif
```

Listing 10. `SSAMOSWAP.W` *for RV64*

Just as for AMOs in the A extension, `SSAMOSWAP.W/D` requires that the address held in `rs1` be naturally aligned to the size of the operand (i.e., eight-byte aligned for *doublewords*, and four-byte aligned for *words*).

The same exception options apply if the address is not naturally aligned.

Just as for AMOs in the A extension, `SSAMOSWAP.W/D` optionally provides release consistency semantics, using the `aq` and `rl` bits, to help implement multiprocessor synchronization. An `SSAMOSWAP.W/D` operation has acquire semantics if `aq=1` and release semantics if `rl=1`.

Stack switching is a common operation in user programs as well as supervisor programs. When a stack switch is performed the stack pointer of the currently active stack is saved into a context data structure and the new stack is made active by loading a new stack pointer from a context data structure.

When shadow stacks are active for a program, the program needs to additionally switch the shadow stack pointer. If the pointer to the top of the deactivated shadow stack is held in a context data structure, then it may be susceptible to memory corruption vulnerabilities. To protect the pointer value, the program may store it at the top of the deactivated shadow stack itself and thereby create a checkpoint. A legal checkpoint is defined as one that holds a value of `X`, where `X` is the address at which the checkpoint is positioned on the shadow stack.

An example sequence to restore the shadow stack pointer from the new shadow stack and save the old shadow stack pointer on the old shadow stack is as follows:

```
# a0 hold pointer to top of new shadow stack to switch to
stack_switch:
   ssrdp ra
   beqz ra, 2f                    # skip if Zicfiss not active
   ssamoswap.d ra, x0,  (a0)      # ra=*[a0] and *[a0]=0
   beq         ra, a0,  1f        # [a0] must be == [ra]
   unimp                          # else crash
1: addi        ra, ra,  XLEN/8    # pop the checkpoint
   csrrw       ra, ssp, ra        # swap ssp: ra=ssp, ssp=ra
   addi        ra, ra,  -(XLEN/8) # checkpoint = "old ssp -
XLEN/8"
   ssamoswap.d x0, ra,  (ra)      # Save checkpoint at "old ssp -
XLEN/8"
2:
```

This sequence uses the `ra` register. If the privilege mode at which this sequence is executed can be interrupted, then the trap handler should save the `ra` on the shadow stack itself. There it is guarded against tampering and can be restored prior to returning from the trap.

When a new shadow stack is created by the supervisor, it needs to store a checkpoint at the highest address on that stack. This enables the shadow stack pointer to be switched using the process outlined in this note. The `SSAMOSWAP.W/D` instruction can be used to store this checkpoint. When the old value at the memory location operated on by `SSAMOSWAP.W/D` is not required, `rd` can be set to `x0`.

Chapter 35. "Zilsd", "Zclsd" Extensions for Load/Store pair for RV32, Version 1.0

The Zilsd & Zclsd extensions provide load/store pair instructions for RV32, reusing the existing RV64 doubleword load/store instruction encodings.

Operands containing `src` for store instructions and `dest` for load instructions are held in aligned `x`-register pairs, i.e., register numbers must be even. Use of misaligned (odd-numbered) registers for these 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 an operand in Zilsd might be held in register `x14`, with bits 63:32 of that operand held in `x15`.

35.1. Load/Store pair instructions (Zilsd)

The Zilsd extension adds the following RV32-only instructions:

RV32	RV64	Mnemonic	Instruction
yes	no	ld rd, offset(rs1)	Load doubleword to register pair, 32-bit encoding
yes	no	sd rs2, offset(rs1)	Store doubleword from register pair, 32-bit encoding

As the access size is 64-bit, accesses are only considered naturally aligned for effective addresses that are a multiple of 8. In this case, these instructions are guaranteed to not raise an address-misaligned exception. Even if naturally aligned, the memory access might not be performed atomically.

If the effective address is a multiple of 4, then each word access is required to be performed atomically.

The following table summarizes the required behavior:

Alignment	Word accesses guaranteed atomic?	Can cause misaligned trap?
8 B	yes	no
4 B not 8 B	yes	yes
else	no	yes

To ensure resumable trap handling is possible for the load instructions, the base register must have its original value if a trap is taken. The other register in the pair can have been updated. This affects x2 for the stack pointer relative instruction and rs1 otherwise.

If an implementation performs a doubleword load access atomically and the register file implements write-back for even/odd register pairs, the mentioned atomicity requirements are inherently fulfilled. Otherwise, an implementation either needs to delay the write-back until the write can be performed atomically, or order sequential writes to the registers to ensure the requirement above is satisfied.

35.2. Compressed Load/Store pair instructions (Zclsd)

Zclsd depends on Zilsd and Zca. It has overlapping encodings with Zcf and is thus incompatible with Zcf.

Zclsd adds the following RV32-only instructions:

RV32	RV64	Mnemonic	Instruction
yes	no	c.ldsp rd, offset(sp)	Stack-pointer based load doubleword to register pair, 16-bit encoding
yes	no	c.sdsp rs2, offset(sp)	Stack-pointer based store doubleword from register pair, 16-bit encoding
yes	no	c.ld rd', offset(rs1')	Load doubleword to register pair, 16-bit encoding
yes	no	c.sd rs2', offset(rs1')	Store doubleword from register pair, 16-bit encoding

35.3. Use of x0 as operand

LD instructions with destination `x0` are processed as any other load, but the result is discarded entirely and x1 is not written. For C.LDSP, usage of `x0` as the destination is reserved.

If using `x0` as `src` of SD or C.SDSP, the entire 64-bit operand is zero — i.e., register `x1` is not accessed.

C.LD and C.SD instructions can only use `x8-15`.

35.4. Exception Handling

For the purposes of RVWMO and exception handling, LD and SD instructions are considered to be misaligned loads and stores, with one additional constraint: an LD or SD instruction whose effective address is a multiple of 4 gives rise to two 4-byte memory operations.

This definition permits LD and SD instructions giving rise to exactly one memory access, regardless of alignment. If instructions with 4-byte-aligned effective address are decomposed into two 32b operations, there is no constraint on the order in which the operations are performed and each operation is guaranteed to be atomic. These decomposed sequences are interruptible. Exceptions might occur on subsequent operations, making the effects of previous operations within the same instruction visible.

Software should make no assumptions about the number or order of accesses these instructions might give rise to, beyond the 4-byte constraint mentioned above. For example, an interrupted store might overwrite the same bytes upon return from the interrupt handler.

35.5. Instructions

35.5.1. ld

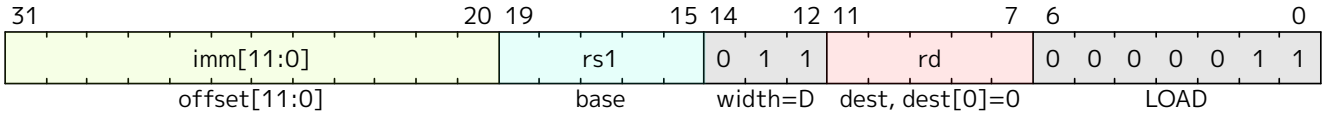
Synopsis

Load doubleword to even/odd register pair, 32-bit encoding

Mnemonic

ld rd, offset(rs1)

Encoding (RV32)



Description

Loads a 64-bit value into registers `rd` and `rd+1`. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

Included in: Zilsd

35.5.2. sd

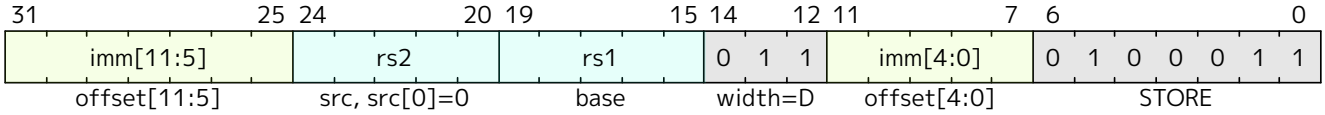
Synopsis

Store doubleword from even/odd register pair, 32-bit encoding

Mnemonic

sd rs2, offset(rs1)

Encoding (RV32)



Description

Stores a 64-bit value from registers `rs2` and `rs2+1`. The effective address is obtained by adding register rs1 to the sign-extended 12-bit offset.

Included in: Zilsd

35.5.3. c.ldsp

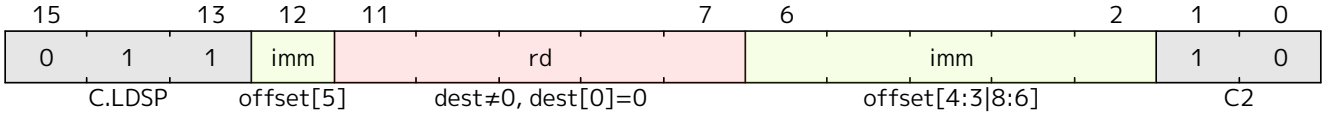
Synopsis

Stack-pointer based load doubleword to even/odd register pair, 16-bit encoding

Mnemonic

c.ldsp rd, offset(sp)

Encoding (RV32)



Description

Loads stack-pointer relative 64-bit value into registers `rd'` and `rd'+1`. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `ld rd, offset(x2)`. C.LDSP is only valid when *rd*≠x0; the code points with *rd*=x0 are reserved.

Included in: Zclsd

35.5.4. c.sdsp

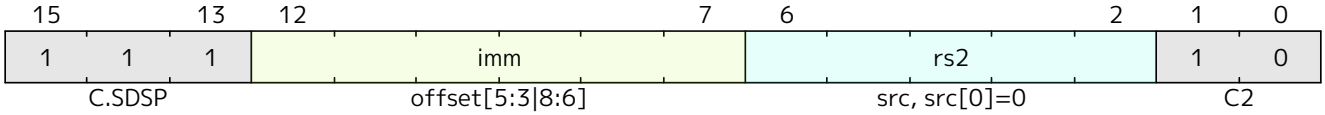
Synopsis

Stack-pointer based store doubleword from even/odd register pair, 16-bit encoding

Mnemonic

c.sdsp rs2, offset(sp)

Encoding (RV32)



Description

Stores a stack-pointer relative 64-bit value from registers `rs2'` and `rs2'+1`. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `sd rs2, offset(x2)`.

Included in: Zclsd

35.5.5. c.ld

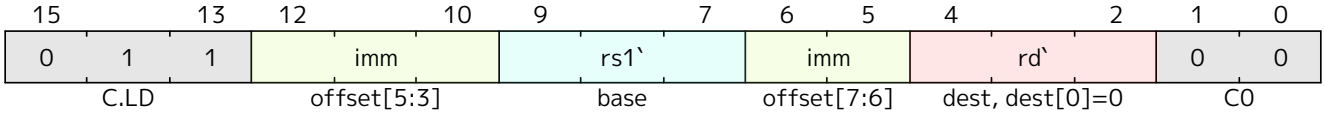
Synopsis

Load doubleword to even/odd register pair, 16-bit encoding

Mnemonic

c.ld rd', offset(rs1')

Encoding (RV32)



Description

Loads a 64-bit value into registers `rd'` and `rd'+1`. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1'.

Included in: Zclsd

35.5.6. c.sd

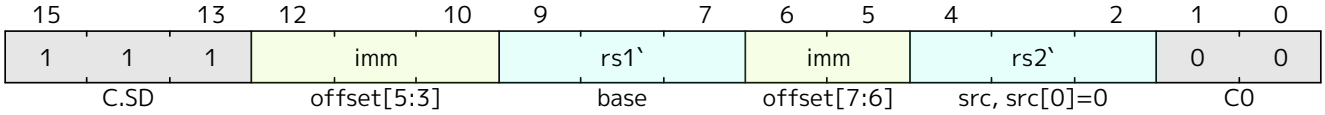
Synopsis

Store doubleword from even/odd register pair, 16-bit encoding

Mnemonic

c.sd rs2', offset(rs1')

Encoding (RV32)



Description

Stores a 64-bit value from registers `rs2'` and `rs2'+1`. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1'. It expands to `sd rs2', offset(rs1')`.

Included in: Zclsd

Chapter 36. RV32/64G Instruction Set Listings

One goal of the RISC-V project is that it be used as a stable software development target. For this purpose, we define a combination of a base ISA (RV32I or RV64I) plus selected standard extensions (IMAFD, Zicsr, Zifencei) as a "general-purpose" ISA, and we use the abbreviation G for the IMAFDZicsr_Zifencei combination of instruction-set extensions. This chapter presents opcode maps and instruction-set listings for RV32G and RV64G.

Table 72. RISC-V base opcode map, inst[1:0]=11

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	*reserved*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	*reserved*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	*custom-2*	*reserved*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	OP-VE	*custom-3*	*reserved*

Table 72 shows a map of the major opcodes for RVG. Opcodes marked as *reserved* should be avoided for custom instruction-set extensions as they might be used by future standard extensions. Major opcodes marked as *custom-0* through *custom-3* will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format.

We believe RV32G and RV64G provide simple but complete instruction sets for a broad range of general-purpose computing. The optional compressed instruction set described in Chapter 28 can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

As we move beyond IMAFDC into further instruction-set extensions, the added instructions tend to be more domain-specific and only provide benefits to a restricted class of applications, e.g., for multimedia or security. Unlike most commercial ISAs, the RISC-V ISA design clearly separates the base ISA and broadly applicable standard extensions from these more specialized additions.

31 27	26	25	24 20	19 15	14 12	11 7	6 0	
funct7			rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]				rs1	funct3	rd	opcode	I-type
imm[11:5]			rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]			rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]						rd	opcode	U-type
imm[20|10:1|11|19:12]						rd	opcode	J-type

RV32I Base Instruction Set						
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL

0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
1000	0011	0011	00000	000	00000	0001111	FENCE.TSO
0000	0001	0000	00000	000	00000	0001111	PAUSE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV64I Base Instruction Set (in addition to RV32I)					
imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32/RV64 *Zifencei* Standard Extension					
imm[11:0]	rs1	001	rd	0001111	FENCE.I

RV32/RV64 *Zicsr* Standard Extension					
csr	rs1	001	rd	1110011	CSRRW
csr	rs1	010	rd	1110011	CSRRS
csr	rs1	011	rd	1110011	CSRRC
csr	uimm	101	rd	1110011	CSRRWI
csr	uimm	110	rd	1110011	CSRRSI
csr	uimm	111	rd	1110011	CSRRCI

RV32M Standard Extension						
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU

0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)						
0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type

RV32A Standard Extension								
00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

RV64A Standard Extension (in addition to RV32A)								
00010	aq	rl	00000	rs1	011	rd	0101111	LR.D
00011	aq	rl	rs2	rs1	011	rd	0101111	SC.D
00001	aq	rl	rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	aq	rl	rs2	rs1	011	rd	0101111	AMOADD.D
00100	aq	rl	rs2	rs1	011	rd	0101111	AMOXOR.D
01100	aq	rl	rs2	rs1	011	rd	0101111	AMOAND.D
01000	aq	rl	rs2	rs1	011	rd	0101111	AMOOR.D
10000	aq	rl	rs2	rs1	011	rd	0101111	AMOMIN.D
10100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAX.D
11000	aq	rl	rs2	rs1	011	rd	0101111	AMOMINU.D
11100	aq	rl	rs2	rs1	011	rd	0101111	AMOMAXU.D

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32F Standard Extension						
imm[11:0]			rs1	010	rd	0000111	FLW
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000		rs2	rs1	rm	rd	1010011	FADD.S
0000100		rs2	rs1	rm	rd	1010011	FSUB.S
0001000		rs2	rs1	rm	rd	1010011	FMUL.S
0001100		rs2	rs1	rm	rd	1010011	FDIV.S
0101100		00000	rs1	rm	rd	1010011	FSQRT.S
0010000		rs2	rs1	000	rd	1010011	FSGNJ.S
0010000		rs2	rs1	001	rd	1010011	FSGNJN.S
0010000		rs2	rs1	010	rd	1010011	FSGNJX.S
0010100		rs2	rs1	000	rd	1010011	FMIN.S
0010100		rs2	rs1	001	rd	1010011	FMAX.S
1100000		00000	rs1	rm	rd	1010011	FCVT.W.S
1100000		00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000		00000	rs1	000	rd	1010011	FMV.X.W
1010000		rs2	rs1	010	rd	1010011	FEQ.S
1010000		rs2	rs1	001	rd	1010011	FLT.S
1010000		rs2	rs1	000	rd	1010011	FLE.S
1110000		00000	rs1	001	rd	1010011	FCLASS.S
1101000		00000	rs1	rm	rd	1010011	FCVT.S.W
1101000		00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000		00000	rs1	000	rd	1010011	FMV.W.X

RV64F Standard Extension (in addition to RV32F)						
1100000	00010	rs1	rm	rd	1010011	FCVT.L.S
1100000	00011	rs1	rm	rd	1010011	FCVT.LU.S
1101000	00010	rs1	rm	rd	1010011	FCVT.S.L
1101000	00011	rs1	rm	rd	1010011	FCVT.S.LU

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32D Standard Extension						
imm[11:0]		rs1	011	rd	0000111	FLD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100111	FSD
rs3	01 rs2	rs1	rm	rd	1000011	FMADD.D
rs3	01 rs2	rs1	rm	rd	1000111	FMSUB.D
rs3	01 rs2	rs1	rm	rd	1001011	FNMSUB.D
rs3	01 rs2	rs1	rm	rd	1001111	FNMADD.D
0000001	rs2	rs1	rm	rd	1010011	FADD.D
0000101	rs2	rs1	rm	rd	1010011	FSUB.D
0001001	rs2	rs1	rm	rd	1010011	FMUL.D
0001101	rs2	rs1	rm	rd	1010011	FDIV.D
0101101	00000	rs1	rm	rd	1010011	FSQRT.D
0010001	rs2	rs1	000	rd	1010011	FSGNJ.D
0010001	rs2	rs1	001	rd	1010011	FSGNJN.D
0010001	rs2	rs1	010	rd	1010011	FSGNJX.D
0010101	rs2	rs1	000	rd	1010011	FMIN.D
0010101	rs2	rs1	001	rd	1010011	FMAX.D
0100000	00001	rs1	rm	rd	1010011	FCVT.S.D
0100001	00000	rs1	rm	rd	1010011	FCVT.D.S
1010001	rs2	rs1	010	rd	1010011	FEQ.D
1010001	rs2	rs1	001	rd	1010011	FLT.D
1010001	rs2	rs1	000	rd	1010011	FLE.D
1110001	00000	rs1	001	rd	1010011	FCLASS.D
1100001	00000	rs1	rm	rd	1010011	FCVT.W.D
1100001	00001	rs1	rm	rd	1010011	FCVT.WU.D
1101001	00000	rs1	rm	rd	1010011	FCVT.D.W
1101001	00001	rs1	rm	rd	1010011	FCVT.D.WU

RV64D Standard Extension (in addition to RV32D)						
1100001	00010	rs1	rm	rd	1010011	FCVT.L.D
1100001	00011	rs1	rm	rd	1010011	FCVT.LU.D
1110001	00000	rs1	000	rd	1010011	FMV.X.D
1101001	00010	rs1	rm	rd	1010011	FCVT.D.L
1101001	00011	rs1	rm	rd	1010011	FCVT.D.LU

1111001	00000	rs1	000	rd	1010011	FMV.D.X

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Q Standard Extension						
imm[11:0]		rs1	100	rd	0000111	FLQ
imm[11:5]	rs2	rs1	100	imm[4:0]	0100111	FSQ
rs3	11 rs2	rs1	rm	rd	1000011	FMADD.Q
rs3	11 rs2	rs1	rm	rd	1000111	FMSUB.Q
rs3	11 rs2	rs1	rm	rd	1001011	FNMSUB.Q
rs3	11 rs2	rs1	rm	rd	1001111	FNMADD.Q
0000011	rs2	rs1	rm	rd	1010011	FADD.Q
0000111	rs2	rs1	rm	rd	1010011	FSUB.Q
0001011	rs2	rs1	rm	rd	1010011	FMUL.Q
0001111	rs2	rs1	rm	rd	1010011	FDIV.Q
0101111	00000	rs1	rm	rd	1010011	FSQRT.Q
0010011	rs2	rs1	000	rd	1010011	FSGNJ.Q
0010011	rs2	rs1	001	rd	1010011	FSGNJN.Q
0010011	rs2	rs1	010	rd	1010011	FSGNJX.Q
0010111	rs2	rs1	000	rd	1010011	FMIN.Q
0010111	rs2	rs1	001	rd	1010011	FMAX.Q
0100000	00011	rs1	rm	rd	1010011	FCVT.S.Q
0100011	00000	rs1	rm	rd	1010011	FCVT.Q.S
0100001	00011	rs1	rm	rd	1010011	FCVT.D.Q
0100011	00001	rs1	rm	rd	1010011	FCVT.Q.D
1010011	rs2	rs1	010	rd	1010011	FEQ.Q
1010011	rs2	rs1	001	rd	1010011	FLT.Q
1010011	rs2	rs1	000	rd	1010011	FLE.Q
1110011	00000	rs1	001	rd	1010011	FCLASS.Q
1100011	00000	rs1	rm	rd	1010011	FCVT.W.Q
1100011	00001	rs1	rm	rd	1010011	FCVT.WU.Q
1101011	00000	rs1	rm	rd	1010011	FCVT.Q.W
1101011	00001	rs1	rm	rd	1010011	FCVT.Q.WU

RV64Q Standard Extension (in addition to RV32Q)						
1100011	00010	rs1	rm	rd	1010011	FCVT.L.Q
1100011	00011	rs1	rm	rd	1010011	FCVT.LU.Q

1101011	00010	rs1	rm	rd	1010011	FCVT.Q.L
1101011	00011	rs1	rm	rd	1010011	FCVT.Q.LU

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Zfh Standard Extension							
imm[11:0]			rs1	001	rd	0000111	FLH
imm[11:5]		rs2	rs1	001	imm[4:0]	0100111	FSH
rs3	10	rs2	rs1	rm	rd	1000011	FMADD.H
rs3	10	rs2	rs1	rm	rd	1000111	FMSUB.H
rs3	10	rs2	rs1	rm	rd	1001011	FNMSUB.H
rs3	10	rs2	rs1	rm	rd	1001111	FNMADD.H
0000010		rs2	rs1	rm	rd	1010011	FADD.H
0000110		rs2	rs1	rm	rd	1010011	FSUB.H
0001010		rs2	rs1	rm	rd	1010011	FMUL.H
0001110		rs2	rs1	rm	rd	1010011	FDIV.H
0101110		00000	rs1	rm	rd	1010011	FSQRT.H
0010010		rs2	rs1	000	rd	1010011	FSGNJ.H
0010010		rs2	rs1	001	rd	1010011	FSGNJN.H
0010010		rs2	rs1	010	rd	1010011	FSGNJX.H
0010110		rs2	rs1	000	rd	1010011	FMIN.H
0010110		rs2	rs1	001	rd	1010011	FMAX.H
0100000		00010	rs1	rm	rd	1010011	FCVT.S.H
0100010		00000	rs1	rm	rd	1010011	FCVT.H.S
0100001		00010	rs1	rm	rd	1010011	FCVT.D.H
0100010		00001	rs1	rm	rd	1010011	FCVT.H.D
0100011		00010	rs1	rm	rd	1010011	FCVT.Q.H
0100010		00011	rs1	rm	rd	1010011	FCVT.H.Q
1010010		rs2	rs1	010	rd	1010011	FEQ.H
1010010		rs2	rs1	001	rd	1010011	FLT.H
1010010		rs2	rs1	000	rd	1010011	FLE.H
1110010		00000	rs1	001	rd	1010011	FCLASS.H
1100010		00000	rs1	rm	rd	1010011	FCVT.W.H
1100010		00001	rs1	rm	rd	1010011	FCVT.WU.H
1110010		00000	rs1	000	rd	1010011	FMV.X.H
1101010		00000	rs1	rm	rd	1010011	FCVT.H.W
1101010		00001	rs1	rm	rd	1010011	FCVT.H.WU
1111010		00000	rs1	000	rd	1010011	FMV.H.X

RV64Zfh Standard Extension (in addition to RV32Zfh)						
1100010	00010	rs1	rm	rd	1010011	FCVT.L.H
1100010	00011	rs1	rm	rd	1010011	FCVT.LU.H
1101010	00010	rs1	rm	rd	1010011	FCVT.H.L
1101010	00011	rs1	rm	rd	1010011	FCVT.H.LU

Zawrs Standard Extension					
000000001101	00000	000	00000	1110011	WRS.NTO
000000011101	00000	000	00000	1110011	WRS.STO

Table 73 lists the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are the only CSRs defined in this specification.

Table 73. RISC-V control and status register (CSR) address map.

Number	Privilege	Name	Description
Floating-Point Control and Status Registers			
0x001	Read write	fflags	Floating-Point Accrued Exceptions.
0x002	Read write	frm	Floating-Point Dynamic Rounding Mode.
0x003	Read write	fcsr	Floating-Point Control and Status Register (frm + fflags).
Counters and Timers			
0xC00	Read-only	cycle	Cycle counter for RDCYCLE instruction.
0xC01	Read-only	time	Timer for RDTIME instruction.
0xC02	Read-only	instret	Instructions-retired counter for RDINSTRET instruction.
0xC80	Read-only	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	Read-only	timeh	Upper 32 bits of time, RV32I only.
0xC82	Read-only	instreth	Upper 32 bits of instret, RV32I only.

Chapter 37. ISA Extension Naming Conventions

This chapter describes the RISC-V ISA extension naming scheme that is used to concisely describe the set of instructions present in a hardware implementation, or the set of instructions used by an application binary interface (ABI).

The RISC-V ISA is designed to support a wide variety of implementations with various experimental instruction-set extensions. We have found that an organized naming scheme simplifies software tools and documentation.

37.1. Case Sensitivity

The ISA naming strings are case insensitive.

37.2. Base Integer ISA

RISC-V ISA strings begin with either RV32I, RV32E, RV64I, or RV64E, indicating the supported address space size in bits for the base integer ISA.

37.3. Instruction-Set Extension Names

Standard ISA extensions are given a name consisting of a single letter. For example, the first four standard extensions to the integer bases are: "M" for integer multiplication and division, "A" for atomic memory instructions, "F" for single-precision floating-point instructions, and "D" for double-precision floating-point instructions. Any RISC-V instruction-set variant can be succinctly described by concatenating the base integer prefix with the names of the included extensions, e.g., "RV64IMAFD".

We have also defined an abbreviation "G" to represent the "IMAFDZicsr_Zifencei" base and extensions, as this is intended to represent our standard general-purpose ISA.

Standard extensions to the RISC-V ISA are given other reserved letters, e.g., "Q" for quad-precision floating-point, or "C" for the 16-bit compressed instruction format.

Some ISA extensions depend on the presence of other extensions, e.g., "D" depends on "F" and "F" depends on "Zicsr". These dependencies may be implicit in the ISA name: for example, RV32IF is equivalent to RV32IFZicsr, and RV32ID is equivalent to RV32IFD and RV32IFDZicsr.

37.4. Underscores

Underscores "_" may be used to separate ISA extensions to improve readability and to provide disambiguation, e.g., "RV32I2_M2_A2".

37.5. Additional Standard Unprivileged Extension Names

Standard unprivileged extensions can also be named by using a single "Z" followed by an alphanumeric name. The name must end with an alphabetical character. The second letter from the end cannot be numeric if the last letter is "p". For example, "Zifencei" names the instruction-fetch fence extension described in Chapter 5.

The first letter following the "Z" conventionally indicates the most closely related alphabetical extension category, IMAFDQLCBKJTPVH. For the "Zfa" extension for additional floating-point instructions, for

example, the letter "f" indicates the extension is related to the "F" standard extension. If multiple "Z" extensions are named, they should be ordered first by category, then alphabetically within a category—for example, "Zicsr_Zifencei_Ztso".

All multi-letter extensions, including those with the "Z" prefix, must be separated from other multi-letter extensions by an underscore, e.g., "RV32IMACZicsr_Zifencei".

37.6. Supervisor-level Instruction-Set Extension Names

Standard extensions that extend the supervisor-level virtual-memory architecture are prefixed with the letters "Sv", followed by an alphanumeric name. Other standard extensions that extend the supervisor-level architecture are prefixed with the letters "Ss", followed by an alphanumeric name. The name must end with an alphabetical character. The second letter from the end cannot be numeric if the last letter is "p". These extensions are further defined in Volume II.

The extensions "sv32", "sv39", "sv48", and "sv59" were defined before the rule against extension names ending in numbers was established.

Standard supervisor-level extensions should be listed after standard unprivileged extensions, and like other multi-letter extensions, must be separated from other multi-letter extensions by an underscore. If multiple supervisor-level extensions are listed, they should be ordered alphabetically.

37.7. Hypervisor-level Instruction-Set Extension Names

Standard extensions that extend the hypervisor-level architecture are prefixed with the letters "Sh". If multiple hypervisor-level extensions are listed, they should be ordered alphabetically.

Many augmentations to the hypervisor-level architecture are more naturally defined as supervisor-level extensions, following the scheme described in the previous section. The "Sh" prefix is used by the few hypervisor-level extensions that have no supervisor-visible effects.

37.8. Machine-level Instruction-Set Extension Names

Standard machine-level instruction-set extensions are prefixed with the letters "Sm".

Standard machine-level extensions should be listed after standard lesser-privileged extensions, and like other multi-letter extensions, must be separated from other multi-letter extensions by an underscore. If multiple machine-level extensions are listed, they should be ordered alphabetically.

37.9. Non-Standard Extension Names

Non-standard extensions are named by using a single "X" followed by the alphanumeric name. The name must end with an alphabetic character. The second letter from the end cannot be numeric if the last letter is "p". For example, "Xhwacha" names the Hwacha vector-fetch ISA extension.

Non-standard extensions must be listed after all standard extensions, and, like other multi-letter extensions, must be separated from other multi-letter extensions by an underscore. For example, an ISA with non-standard extensions Argle and Bargle may be named "RV64IZifencei_Xargle_Xbargle".

If multiple non-standard extensions are listed, they should be ordered alphabetically. Like other multi-letter extensions, they should be separated from other multi-letter extensions by an underscore.

37.10. Version Numbers

Recognizing that instruction sets may expand or alter over time, we encode extension version numbers following the extension name. Version numbers are divided into major and minor version numbers, separated by a "p". If the minor version is "0", then "p0" can be omitted from the version string. To avoid ambiguity, no extension name may end with a number or a "p" preceded by a number.

Because the "P" extension for Packed SIMD can be confused for the decimal point in a version number, it must be preceded by an underscore if it follows another extension with a version number. For example, "rv32i2p2" means version 2.2 of RV32I, whereas "rv32i2_p2" means version 2.0 of RV32I with version 2.0 of the P extension.

Changes in major version numbers imply a loss of backwards compatibility, whereas changes in only the minor version number must be backwards-compatible. For example, the original 64-bit standard ISA defined in release 1.0 of this manual can be written in full as "RV64I1p0M1p0A1p0F1p0D1p0", more concisely as "RV64I1M1A1F1D1".

We introduced the version numbering scheme with the second release. Hence, we define the default version of a standard extension to be the version present at that time, e.g., "RV32I" is equivalent to "RV32I2".

37.11. Subset Naming Convention

Table 74 summarizes the standardized extension names. The table also defines the canonical order in which extension names must appear in the name string, with top-to-bottom in table indicating first-to-last in the name string, e.g., RV32IMACV is legal, whereas RV32IMAVC is not.

Table 74. Standard ISA extension names.

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	Zmmul
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General	G	IMAFDZicsr_Zifencei
Quad-Precision Floating-Point	Q	D
16-bit Compressed Instructions	C	
B Extension	B	
Packed-SIMD Extensions	P	
Vector Extension	V	D
Hypervisor Extension	H	
Additional Standard Unprivileged Extensions		
Additional Standard unprivileged extensions "abc"	Zabc	
Standard Supervisor-Level Extensions		
Supervisor-level extension "def"	Ssdef	

Subset	Name	Implies
Standard Hypervisor-Level Extensions		
Hypervisor-level extension "ghi"	Shghi	
Standard Machine-Level Extensions		
Machine-level extension "jkl"	Smjkl	
Non-Standard Extensions		
Non-standard extension "mno"	Xmno	

Appendix A: RVWMO Explanatory Material, Version 0.1

This section provides more explanation for RVWMO Chapter 18, using more informal language and concrete examples. These are intended to clarify the meaning and intent of the axioms and preserved program order rules. This appendix should be treated as commentary; all normative material is provided in Chapter 18 and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in Section A.7. Any other discrepancies are unintentional.

A.1. Why RVWMO?

Memory consistency models fall along a loose spectrum from weak to strong. Weak memory models allow more hardware implementation flexibility and deliver arguably better performance, performance per watt, power, scalability, and hardware verification overheads than strong models, at the expense of a more complex programming model. Strong models provide simpler programming models, but at the cost of imposing more restrictions on the kinds of (non-speculative) hardware optimizations that can be performed in the pipeline and in the memory system, and in turn imposing some cost in terms of power, area overhead, and verification burden.

RISC-V has chosen the RVWMO memory model, a variant of release consistency. This places it in between the two extremes of the memory model spectrum. The RVWMO memory model enables architects to build simple implementations, aggressive implementations, implementations embedded deeply inside a much larger system and subject to complex memory system interactions, or any number of other possibilities, all while simultaneously being strong enough to support programming language memory models at high performance.

To facilitate the porting of code from other architectures, some hardware implementations may choose to implement the Ztso extension, which provides stricter RVTSO ordering semantics by default. Code written for RVWMO is automatically and inherently compatible with RVTSO, but code written assuming RVTSO is not guaranteed to run correctly on RVWMO implementations. In fact, most RVWMO implementations will (and should) simply refuse to run RVTSO-only binaries. Each implementation must therefore choose whether to prioritize compatibility with RVTSO code (e.g., to facilitate porting from x86) or whether to instead prioritize compatibility with other RISC-V cores implementing RVWMO.

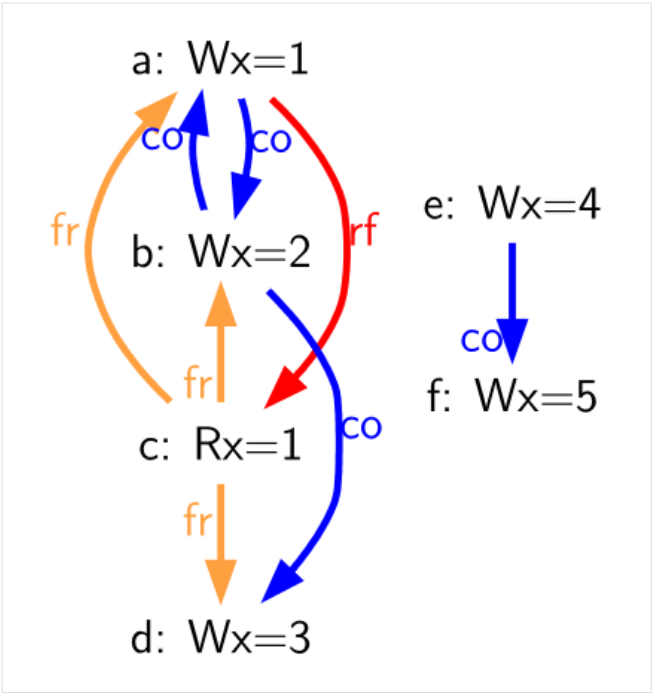
Some fences and/or memory ordering annotations in code written for RVWMO may become redundant under RVTSO; the cost that the default of RVWMO imposes on Ztso implementations is the incremental overhead of fetching those fences (e.g., FENCE R,RW and FENCE RW,W) which become no-ops on that implementation. However, these fences must remain present in the code if compatibility with non-Ztso implementations is desired.

A.2. Litmus Tests

The explanations in this chapter make use of *litmus tests*, or small programs designed to test or highlight one particular aspect of a memory model. Litmus sample shows an example of a litmus test with two harts. As a convention for this figure and for all figures that follow in this chapter, we assume that `s0-s2` are pre-set to the same value in all harts and that `s0` holds the address labeled `x`, `s1` holds `y`, and `s2` holds `z`, where `x`, `y`, and `z` are disjoint memory locations aligned to 8 byte boundaries. All other registers and all referenced memory locations are presumed to be initialized to zero. Each figure shows the litmus test code on the left, and a visualization of one particular valid or invalid execution on the right.

Table 75. A sample litmus test and one forbidden execution (`a0=1`).

	Hart 0		Hart 1
	⋮		⋮
	li t1,1		li t4,4
(a)	sw t1,0(s0)	(e)	sw t4,0(s0)
	⋮		⋮
	li t2,2		
(b)	sw t2,0(s0)		
	⋮		⋮
(c)	lw a0,0(s0)		
	⋮		⋮
	li t3,3		li t5,5
(d)	sw t3,0(s0)	(f)	sw t5,0(s0)
	⋮		⋮



Litmus tests are used to understand the implications of the memory model in specific concrete situations. For example, in the litmus test of Litmus sample, the final value of `a0` in the first hart can be either 2, 4, or 5, depending on the dynamic interleaving of the instruction stream from each hart at runtime. However, in this example, the final value of `a0` in Hart 0 will never be 1 or 3; intuitively, the value 1 will no longer be visible at the time the load executes, and the value 3 will not yet be visible by the time the load executes. We analyze this test and many others below.

Table 76. A key for the litmus test diagrams drawn in this appendix

Edge	Full Name (and explanation)
rf	Reads From (from each store to the loads that return a value written by that store)
co	Coherence (a total order on the stores to each address)
fr	From-Reads (from each load to co-successors of the store from which the load returned a value)
ppo	Preserved Program Order
fence	Orderings enforced by a FENCE instruction
addr	Address Dependency
ctrl	Control Dependency
data	Data Dependency

The diagram shown to the right of each litmus test shows a visual representation of the particular execution candidate being considered. These diagrams use a notation that is common in the memory model literature for constraining the set of possible global memory orders that could produce the execution in question. It is also the basis for the *herd* models presented in Section B.2. This notation is explained in Table 76. Of the listed relations, rf edges between harts, co edges, fr edges, and ppo edges directly constrain the global memory order (as do fence, addr, data, and some ctrl edges, via ppo). Other edges (such as intra-hart rf edges) are informative but do not constrain the global memory order.

For example, in Litmus sample, `a0=1` could occur only if one of the following were true:

- (b) appears before (a) in global memory order (and in the coherence order co). However, this violates RVWMO PPO rule `ppo:→st`. The co edge from (b) to (a) highlights this contradiction.
- (a) appears before (b) in global memory order (and in the coherence order co). However, in this case, the Load Value Axiom would be violated, because (a) is not the latest matching store prior to (c) in program order. The fr edge from (c) to (b) highlights this contradiction.

Since neither of these scenarios satisfies the RVWMO axioms, the outcome `a0=1` is forbidden.

Beyond what is described in this appendix, a suite of more than seven thousand litmus tests is available at github.com/litmus-tests/litmus-tests-riscv.

The litmus tests repository also provides instructions on how to run the litmus tests on RISC-V hardware and how to compare the results with the operational and axiomatic models.

In the future, we expect to adapt these memory model litmus tests for use as part of the RISC-V compliance test suite as well.

A.3. Explaining the RVWMO Rules

In this section, we provide explanation and examples for all of the RVWMO rules and axioms.

A.3.1. Preserved Program Order and Global Memory Order

Preserved program order represents the subset of program order that must be respected within the global memory order. Conceptually, events from the same hart that are ordered by preserved program order must appear in that order from the perspective of other harts and/or observers. Events from the same hart that are not ordered by preserved program order, on the other hand, may appear reordered from the perspective of other harts and/or observers.

Informally, the global memory order represents the order in which loads and stores perform. The formal memory model literature has moved away from specifications built around the concept of performing, but the idea is still useful for building up informal intuition. A load is said to have performed when its return value is determined. A store is said to have performed not when it has executed inside the pipeline, but rather only when its value has been propagated to globally visible memory. In this sense, the global memory order also represents the contribution of the coherence protocol and/or the rest of the memory system to interleave the (possibly reordered) memory accesses being issued by each hart into a single total order agreed upon by all harts.

The order in which loads perform does not always directly correspond to the relative age of the values those two loads return. In particular, a load *b* may perform before another load *a* to the same address (i.e., *b* may execute before *a*, and *b* may appear before *a* in the global memory order), but *a* may nevertheless return an older value than *b*. This discrepancy captures (among other things) the reordering effects of buffering placed between the core and memory. For example, *b* may have returned a value from a store in the store buffer, while *a* may have ignored that younger store and read an older value from memory instead. To account for this, at the time each load performs, the value it returns is determined by the load value axiom, not just strictly by determining the most recent store to the same address in the global memory order, as described below.

A.3.2. Load value axiom

Section 18.1.4.1: Each byte of each load i returns the value written to that byte by the store that is the latest in global memory order among the following stores:

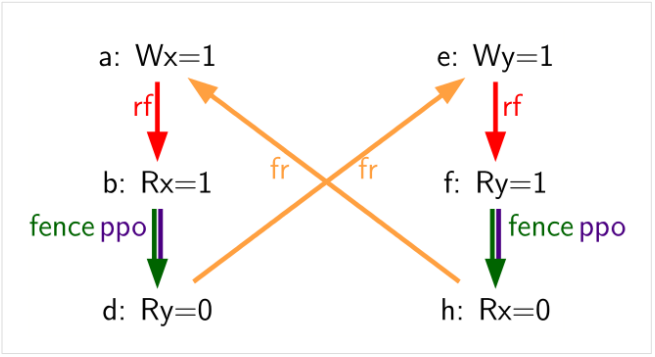
1. *Stores that write that byte and that precede i in the global memory order*
2. *Stores that write that byte and that precede i in program order*

Preserved program order is *not* required to respect the ordering of a store followed by a load to an overlapping address. This complexity arises due to the ubiquity of store buffers in nearly all implementations. Informally, the load may perform (return a value) by forwarding from the store while the store is still in the store buffer, and hence before the store itself performs (writes back to globally visible memory). Any other hart will therefore observe the load as performing before the store.

Consider the Table 77. When running this program on an implementation with store buffers, it is possible to arrive at the final outcome `a0=1, a1=0, a2=1, a3=0` as follows:

Table 77. A store buffer forwarding litmus test (outcome permitted)

Hart 0	Hart 1
li t1, 1	li t1, 1
(a) sw t1,0(s0)	(e) sw t1,0(s1)
(b) lw a0,0(s0)	(f) lw a2,0(s1)
(c) fence r,r	(g) fence r,r
(d) lw a1,0(s1)	(h) lw a3,0(s0)
Outcome: `a0=1`, `a1=0`, `a2=1`, `a3=0`	



- (a) executes and enters the first hart's private store buffer
- (b) executes and forwards its return value 1 from (a) in the store buffer
- (c) executes since all previous loads (i.e., (b)) have completed
- (d) executes and reads the value 0 from memory

- (e) executes and enters the second hart's private store buffer
- (f) executes and forwards its return value 1 from (e) in the store buffer
- (g) executes since all previous loads (i.e., (f)) have completed
- (h) executes and reads the value 0 from memory
- (a) drains from the first hart's store buffer to memory
- (e) drains from the second hart's store buffer to memory

Therefore, the memory model must be able to account for this behavior.

To put it another way, suppose the definition of preserved program order did include the following hypothetical rule: memory access *a* precedes memory access *b* in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program order and *a* and *b* are accesses to the same memory location, *a* is a write, and *b* is a read. Call this "Rule X". Then we get the following:

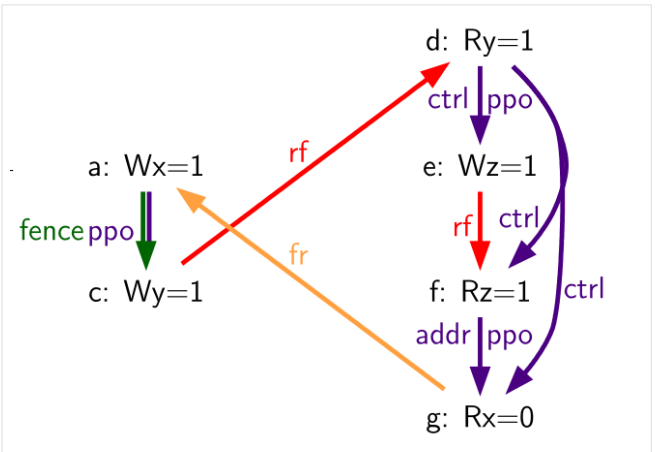
- (a) precedes (b): by rule X
- (b) precedes (d): by rule 4
- (d) precedes (e): by the load value axiom. Otherwise, if (e) preceded (d), then (d) would be required to return the value 1. (This is a perfectly legal execution; it's just not the one in question)
- (e) precedes (f): by rule X
- (f) precedes (h): by rule 4
- (h) precedes (a): by the load value axiom, as above.

The global memory order must be a total order and cannot be cyclic, because a cycle would imply that every event in the cycle happens before itself, which is impossible. Therefore, the execution proposed above would be forbidden, and hence the addition of rule X would forbid implementations with store buffer forwarding, which would clearly be undesirable.

Nevertheless, even if (b) precedes (a) and/or (f) precedes (e) in the global memory order, the only sensible possibility in this example is for (b) to return the value written by (a), and likewise for (f) and (e). This combination of circumstances is what leads to the second option in the definition of the load value axiom. Even though (b) precedes (a) in the global memory order, (a) will still be visible to (b) by virtue of sitting in the store buffer at the time (b) executes. Therefore, even if (b) precedes (a) in the global memory order, (b) should return the value written by (a) because (a) precedes (b) in program order. Likewise for (e) and (f).

Table 78. The "PPOCA" store buffer forwarding litmus test (outcome permitted)

Hart 0		Hart 1	
	li t1, 1		li t1, 1
(a)	sw t1,0(s0)		LOOP:
(b)	fence w,w	(d)	lw a0,0(s1)
(c)	sw t1,0(s1)		beqz a0, LOOP
		(e)	sw t1,0(s2)
		(f)	lw a1,0(s2)
			xor a2,a1,a1
			add s0,s0,a2
		(g)	lw a2,0(s0)
Outcome: `a0=1`, `a1=1`, `a2=0`			



Another test that highlights the behavior of store buffers is shown in Table 78. In this example, (d) is ordered before (e) because of the control dependency, and (f) is ordered before (g) because of the address dependency. However, (e) is *not* necessarily ordered before (f), even though (f) returns the value written by (e). This could correspond to the following sequence of events:

- (e) executes speculatively and enters the second hart's private store buffer (but does not drain to memory)
- (f) executes speculatively and forwards its return value 1 from (e) in the store buffer
- (g) executes speculatively and reads the value 0 from memory
- (a) executes, enters the first hart's private store buffer, and drains to memory
- (b) executes and retires
- (c) executes, enters the first hart's private store buffer, and drains to memory
- (d) executes and reads the value 1 from memory
- (e), (f), and (g) commit, since the speculation turned out to be correct
- (e) drains from the store buffer to memory

A.3.3. Atomicity axiom

Atomicity Axiom (for Aligned Atomics): If r and w are paired load and store operations generated by aligned LR and SC instructions in a hart h, s is a store to byte x, and r returns a value written by s, then s must precede w in the global memory order, and there can be no store from a hart other than h to byte x following s and preceding w in the global memory order.

The RISC-V architecture decouples the notion of atomicity from the notion of ordering. Unlike architectures such as TSO, RISC-V atomics under RVWMO do not impose any ordering requirements by default. Ordering semantics are only guaranteed by the PPO rules that otherwise apply.

RISC-V contains two types of atomics: AMOs and LR/SC pairs. These conceptually behave differently, in the following way. LR/SC behave as if the old value is brought up to the core, modified, and written back to memory, all while a reservation is held on that memory location. AMOs on the other hand conceptually behave as if they are performed directly in memory. AMOs are therefore inherently atomic, while LR/SC pairs are atomic in the slightly different sense that the memory location in question will not be modified by another hart during the time the original hart holds the reservation.

Table 79. In all four (independent) instances, the final store-conditional instruction is permitted but not guaranteed to succeed.

(a) lr.d a0, 0(s0)	(a) lr.d a0, 0(s0)	(a) lr.w a0, 0(s0)	(a) lr.w a0, 0(s0)
(b) sd t1, 0(s0)	(b) sw t1, 4(s0)	(b) sw t1, 4(s0)	(b) sw t1, 4(s0)
(c) sc.d t3, t2, 0(s0)	(c) sc.d t3, t2, 0(s0)	(c) sc.w t3, t2, 0(s0)	(c) addi s0, s0, 8
			(d) sc.w t3, t2, 0(s0)

The atomicity axiom forbids stores from other harts from being interleaved in global memory order between an LR and the SC paired with that LR. The atomicity axiom does not forbid loads from being interleaved between the paired operations in program order or in the global memory order, nor does it forbid stores from the same hart or stores to non-overlapping locations from appearing between the paired operations in either program order or in the global memory order. For example, the SC instructions in Table 79 may (but are not guaranteed to) succeed. None of those successes would violate the atomicity axiom, because the intervening non-conditional stores are from the same hart as the paired load-reserved and store-conditional instructions. This way, a memory system that tracks memory accesses at cache line

granularity (and which therefore will see the four snippets of Table 79 as identical) will not be forced to fail a store-conditional instruction that happens to (falsely) share another portion of the same cache line as the memory location being held by the reservation.

The atomicity axiom also technically supports cases in which the LR and SC touch different addresses and/or use different access sizes; however, use cases for such behaviors are expected to be rare in practice. Likewise, scenarios in which stores from the same hart between an LR/SC pair actually overlap the memory location(s) referenced by the LR or SC are expected to be rare compared to scenarios where the intervening store may simply fall onto the same cache line.

A.3.4. Progress axiom

Progress Axiom: No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.

The progress axiom ensures a minimal forward progress guarantee. It ensures that stores from one hart will eventually be made visible to other harts in the system in a finite amount of time, and that loads from other harts will eventually be able to read those values (or successors thereof). Without this rule, it would be legal, for example, for a spinlock to spin infinitely on a value, even with a store from another hart unlocking the spinlock.

The progress axiom is intended not to impose any other notion of fairness, latency, or quality of service onto the harts in a RISC-V implementation. Any stronger notions of fairness are up to the rest of the ISA and/or up to the platform and/or device to define and implement.

The forward progress axiom will in almost all cases be naturally satisfied by any standard cache coherence protocol. Implementations with non-coherent caches may have to provide some other mechanism to ensure the eventual visibility of all stores (or successors thereof) to all harts.

A.3.5. Overlapping-Address Orderings (Rules 1-3)

Rule 1: b is a store, and a and b access overlapping memory addresses

Rule 2: a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations

Rule 3: a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

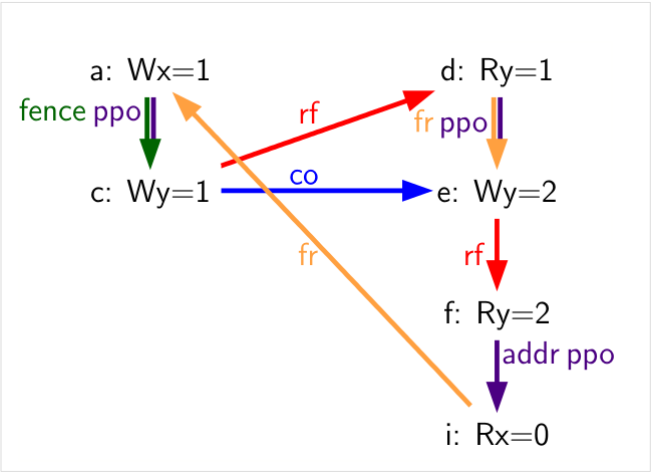
Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model. Same-address orderings from a store to a later load, on the other hand, do not need to be enforced. As discussed in Load value axiom, this reflects the observable behavior of implementations that forward values from buffered stores to later loads.

Same-address load-load ordering requirements are far more subtle. The basic requirement is that a younger load must not return a value that is older than a value returned by an older load in the same hart to the same address. This is often known as "CoRR" (Coherence for Read-Read pairs), or as part of a broader "coherence" or "sequential consistency per location" requirement. Some architectures in the past have relaxed same-address load-load ordering, but in hindsight this is generally considered to complicate the programming model too much, and so RVWMO requires CoRR ordering to be enforced. However, because the global memory order corresponds to the order in which loads perform rather than the ordering of the values being returned, capturing CoRR requirements in terms of the global memory order requires a bit of

indirection.

Table 80. Litmus test MP+fence.w.w+fre-rfi-addr (outcome permitted)

Hart 0		Hart 1	
	li t1, 1		li t2, 2
(a)	sw t1,0(s0)	(d)	lw a0,0(s1)
(b)	fence w, w	(e)	sw t2,0(s1)
(c)	sw t1,0(s1)	(f)	lw a1,0(s1)
		(g)	xor t3,a1,a1
		(h)	add s0,s0,t3
		(i)	lw a2,0(s0)
Outcome: `a0=1`, `a1=2`, `a2=0`			



Consider the litmus test of Table 80, which is one particular instance of the more general "fri-rfi" pattern. The term "fri-rfi" refers to the sequence (d), (e), (f): (d) "from-reads" (i.e., reads from an earlier write than) (e) which is the same hart, and (f) reads from (e) which is in the same hart.

From a microarchitectural perspective, outcome `a0=1`, `a1=2`, `a2=0` is legal (as are various other less subtle outcomes). Intuitively, the following would produce the outcome in question:

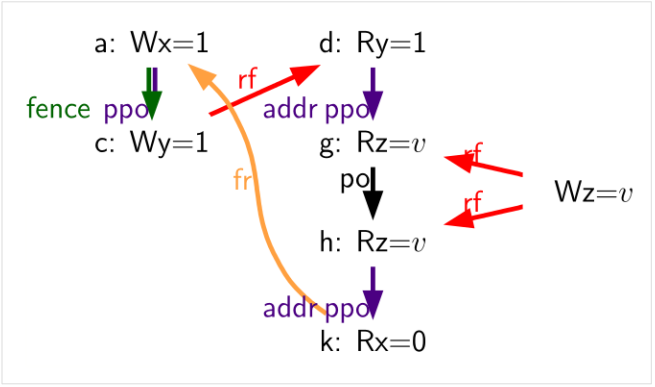
- (d) stalls (for whatever reason; perhaps it's stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer (but does not yet drain to memory)
- (f) executes and forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (a) executes and drains to memory, (b) executes, and (c) executes and drains to memory
- (d) unstalls and executes
- (e) drains from the store buffer to memory

This corresponds to a global memory order of (f), (i), (a), (c), (d), (e). Note that even though (f) performs before (d), the value returned by (f) is newer than the value returned by (d). Therefore, this execution is legal and does not violate the CoRR requirements.

Likewise, if two back-to-back loads return the values written by the same store, then they may also appear out-of-order in the global memory order without violating CoRR. Note that this is not the same as saying that the two loads return the same value, since two different stores may write the same value.

Table 81. Litmus test RSW (outcome permitted)

Hart 0		Hart 1	
li t1, 1		(d)	lw a0,0(s1)
(a)	sw t1,0(s0)	(e)	xor t2,a0,a0
(b)	fence w, w	(f)	add s4,s2,t2
(c)	sw t1,0(s1)	(g)	lw a1,0(s4)
		(h)	lw a2,0(s2)
		(i)	xor t3,a2,a2
		(j)	add s0,s0,t3
		(k)	lw a3,0(s0)
Outcome: `a0=1`, `a1=v`, `a2=v`, `a3=0`			



Consider the litmus test of Table 81. The outcome `a0=1`, `a1=v`, `a2=v`, `a3=0` (where *v* is some value written by another hart) can be observed by allowing (g) and (h) to be reordered. This might be done speculatively, and the speculation can be justified by the microarchitecture (e.g., by snooping for cache invalidations and finding none) because replaying (h) after (g) would return the value written by the same store anyway. Hence assuming `a1` and `a2` would end up with the same value written by the same store anyway, (g) and (h) can be legally reordered. The global memory order corresponding to this execution would be (h),(k),(a),(c),(d),(g).

Executions of the test in Table 81 in which `a1` does not equal `a2` do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in that case result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, rule 2 forbids this CoRR violation from occurring. As such, rule 2 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

There is one more overlapping-address rule: rule 3 simply states that a value cannot be returned from an AMO or SC to a subsequent load until the AMO or SC has (in the case of the SC, successfully) performed globally. This follows somewhat naturally from the conceptual view that both AMOs and SC instructions are meant to be performed atomically in memory. However, notably, rule 3 states that hardware may not even non-speculatively forward the value being stored by an AMOSWAP to a subsequent load, even though for AMOSWAP that store value is not actually semantically dependent on the previous value in memory, as is the case for the other AMOs. The same holds true even when forwarding from SC store values that are not semantically dependent on the value returned by the paired LR.

The three PPO rules above also apply when the memory accesses in question only overlap partially. This can occur, for example, when accesses of different sizes are used to access the same object. Note also that the base addresses of two overlapping memory operations need not necessarily be the same for two

memory accesses to overlap. When misaligned memory accesses are being used, the overlapping-address PPO rules apply to each of the component memory accesses independently.

A.3.6. Fences (Rule 4)

Rule 4: There is a FENCE instruction that orders a before b

By default, the FENCE instruction ensures that all memory accesses from instructions preceding the fence in program order (the "predecessor set") appear earlier in the global memory order than memory accesses from instructions appearing after the fence in program order (the "successor set"). However, fences can optionally further restrict the predecessor set and/or the successor set to a smaller set of memory accesses in order to provide some speedup. Specifically, fences have PR, PW, SR, and SW bits which restrict the predecessor and/or successor sets. The predecessor set includes loads (resp.stores) if and only if PR (resp.PW) is set. Similarly, the successor set includes loads (resp.stores) if and only if SR (resp.SW) is set.

The FENCE encoding currently has nine non-trivial combinations of the four bits PR, PW, SR, and SW, plus one extra encoding FENCE.TSO which facilitates mapping of "acquire+release" or RVTSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, six are commonly used in practice:

- FENCE RW,RW
- FENCE.TSO
- FENCE RW,W
- FENCE R,RW
- FENCE R,R
- FENCE W,W

FENCE instructions using other combinations of PR, PW, SR, and SW are not normally used in the Linux or C++ memory models but are otherwise well defined.

Finally, we note that since RISC-V uses a multi-copy atomic memory model, programmers can reason about fences bits in a thread-local manner. Fences in RISC-V are not cumulative, as they are in some non-multi-copy-atomic memory models.

A.3.7. Explicit Synchronization (Rules 5-8)

Rule 5: a has an acquire annotation

Rule 6: b has a release annotation

Rule 7: a and b both have RCsc annotations

Rule 8: a is paired with b

An *acquire* operation, as would be used at the start of a critical section, requires all memory operations following the acquire in program order to also follow the acquire in the global memory order. This ensures, for example, that all loads and stores inside the critical section are up to date with respect to the synchronization variable being used to protect it. Acquire ordering can be enforced in one of two ways: with an acquire annotation, which enforces ordering with respect to just the synchronization variable itself, or with a FENCE R,RW, which enforces ordering with respect to all previous loads.

```
        sd          x1, (a1)     # Arbitrary unrelated store
        ld          x2, (a2)     # Arbitrary unrelated load
        li          t0, 1        # Initialize swap value.
    again:
        amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
        bnez        t0, again    # Retry if held.
        # ...
        # Critical section.
        # ...
        amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
        sd          x3, (a3)     # Arbitrary unrelated store
        ld          x4, (a4)     # Arbitrary unrelated load
```

Listing 11. A spinlock with atomics

Consider Example 1. Because this example uses *aq*, the loads and stores in the critical section are guaranteed to appear in the global memory order after the AMOSWAP used to acquire the lock. However, assuming `a0`, `a1`, and `a2` point to different memory locations, the loads and stores in the critical section may or may not appear after the "Arbitrary unrelated load" at the beginning of the example in the global memory order.

```
        sd          x1, (a1)     # Arbitrary unrelated store
        ld          x2, (a2)     # Arbitrary unrelated load
        li          t0, 1        # Initialize swap value.
    again:
        amoswap.w   t0, t0, (a0) # Attempt to acquire lock.
        fence       r, rw        # Enforce "acquire" memory ordering
        bnez        t0, again    # Retry if held.
        # ...
        # Critical section.
        # ...
        fence       rw, w        # Enforce "release" memory ordering
        amoswap.w   x0, x0, (a0) # Release lock by storing 0.
        sd          x3, (a3)     # Arbitrary unrelated store
        ld          x4, (a4)     # Arbitrary unrelated load
```

Listing 12. A spinlock with fences

Now, consider the alternative in Example 2. In this case, even though the AMOSWAP does not enforce ordering with an *aq* bit, the fence nevertheless enforces that the acquire AMOSWAP appears earlier in the global memory order than all loads and stores in the critical section. Note, however, that in this case, the fence also enforces additional orderings: it also requires that the "Arbitrary unrelated load" at the start of the program appears earlier in the global memory order than the loads and stores of the critical section. (This particular fence does not, however, enforce any ordering with respect to the "Arbitrary unrelated store" at the start of the snippet.) In this way, fence-enforced orderings are slightly coarser than orderings enforced by *.aq*.

Release orderings work exactly the same as acquire orderings, just in the opposite direction. Release semantics require all loads and stores preceding the release operation in program order to also precede the release operation in the global memory order. This ensures, for example, that memory accesses in a critical section appear before the lock-releasing store in the global memory order. Just as for acquire semantics,

release semantics can be enforced using release annotations or with a FENCE RW,W operation. Using the same examples, the ordering between the loads and stores in the critical section and the "Arbitrary unrelated store" at the end of the code snippet is enforced only by the FENCE RW,W in Example 2, not by the *rl* in Example 1.

With RCpc annotations alone, store-release-to-load-acquire ordering is not enforced. This facilitates the porting of code written under the TSO and/or RCpc memory models. To enforce store-release-to-load-acquire ordering, the code must use store-release-RCsc and load-acquire-RCsc operations so that PPO rule 7 applies. RCpc alone is sufficient for many use cases in C/C++ but is insufficient for many other use cases in C/C++, Java, and Linux, to name just a few examples; see Memory Porting for details.

PPO rule 8 indicates that an SC must appear after its paired LR in the global memory order. This will follow naturally from the common use of LR/SC to perform an atomic read-modify-write operation due to the inherent data dependency. However, PPO rule 8 also applies even when the value being stored does not syntactically depend on the value returned by the paired LR.

Lastly, we note that, as with fences, ordering annotations are not cumulative.

A.3.8. Syntactic Dependencies (Rules 9-11)

Rule 9: b has a syntactic address dependency on a

Rule 10: b has a syntactic data dependency on a

Rule 11: b is a store, and b has a syntactic control dependency on a

Dependencies from a load to a later memory operation in the same hart are respected by the RVWMO memory model. The Alpha memory model was notable for choosing *not* to enforce the ordering of such dependencies, but most modern hardware and software memory models consider allowing dependent instructions to be reordered too confusing and counterintuitive. Furthermore, modern code sometimes intentionally uses such dependencies as a particularly lightweight ordering enforcement mechanism.

The terms in Section 18.1.2 work as follows. Instructions are said to carry dependencies from their source register(s) to their destination register(s) whenever the value written into each destination register is a function of the source register(s). For most instructions, this means that the destination register(s) carry a dependency from all source register(s). However, there are a few notable exceptions. In the case of memory instructions, the value written into the destination register ultimately comes from the memory system rather than from the source register(s) directly, and so this breaks the chain of dependencies carried from the source register(s). In the case of unconditional jumps, the value written into the destination register comes from the current `pc` (which is never considered a source register by the memory model), and so likewise, JALR (the only jump with a source register) does not carry a dependency from *rs1* to *rd*.

```
1 (a) fadd f3,f1,f2
2 (b) fadd f6,f4,f5
3 (c) csrrs a0,fflags,x0
```

Listing 13. (c) has a syntactic dependency on both (a) and (b) via fflags, a destination register that both (a) and (b) implicitly accumulate into

The notion of accumulating into a destination register rather than writing into it reflects the behavior of CSRs such as `fflags`. In particular, an accumulation into a register does not clobber any previous writes or accumulations into the same register. For example, in Listing 13, (c) has a syntactic dependency on both (a) and (b).

Like other modern memory models, the RVWMO memory model uses syntactic rather than semantic dependencies. In other words, this definition depends on the identities of the registers being accessed by different instructions, not the actual contents of those registers. This means that an address, control, or data dependency must be enforced even if the calculation could seemingly be `optimized away`. This choice ensures that RVWMO remains compatible with code that uses these false syntactic dependencies as a lightweight ordering mechanism.

```
1 ld a1,0(s0)
2 xor a2,a1,a1
3 add s1,s1,a2
4 ld a5,0(s1)
```

Listing 14. A syntactic address dependency

For example, there is a syntactic address dependency from the memory operation generated by the first instruction to the memory operation generated by the last instruction in Listing 14, even though `a1` XOR `a1` is zero and hence has no effect on the address accessed by the second load.

The benefit of using dependencies as a lightweight synchronization mechanism is that the ordering enforcement requirement is limited only to the specific two instructions in question. Other non-dependent instructions may be freely reordered by aggressive implementations. One alternative would be to use a load-acquire, but this would enforce ordering for the first load with respect to *all* subsequent instructions. Another would be to use a FENCE R,R, but this would include all previous and all subsequent loads, making this option more expensive.

```
1 lw x1,0(x2)
2 bne x1,x0,next
3 sw x3,0(x4)
4 next: sw x5,0(x6)
```

Listing 15. A syntactic control dependency

Control dependencies behave differently from address and data dependencies in the sense that a control dependency always extends to all instructions following the original target in program order. Consider Listing 15 the instruction at `next` will always execute, but the memory operation generated by that last instruction nevertheless still has a control dependency from the memory operation generated by the first instruction.

```
1 lw x1,0(x2)
2 bne x1,x0,next
3 next: sw x3,0(x4)
```

Listing 16. Another syntactic control dependency

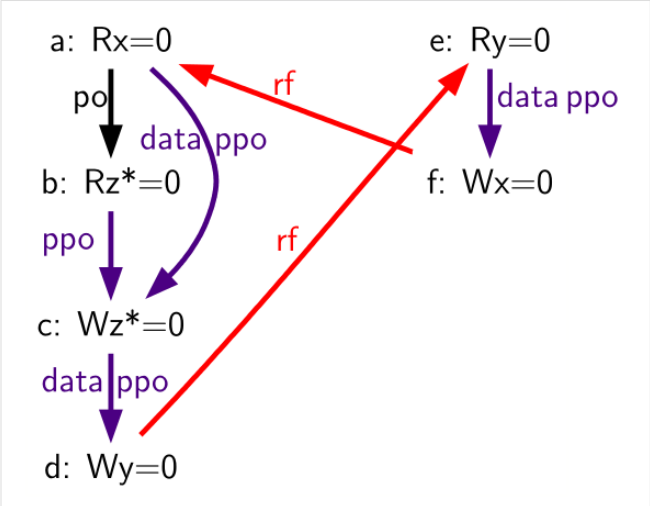
Likewise, consider Listing 16. Even though both branch outcomes have the same target, there is still a control dependency from the memory operation generated by the first instruction in this snippet to the memory operation generated by the last instruction. This definition of control dependency is subtly stronger than what might be seen in other contexts (e.g., C++), but it conforms with standard definitions of control dependencies in the literature.

Notably, PPO rules 9-11 are also intentionally designed to respect dependencies that originate from the output of a successful store-conditional instruction. Typically, an SC instruction will be followed by a

conditional branch checking whether the outcome was successful; this implies that there will be a control dependency from the store operation generated by the SC instruction to any memory operations following the branch. PPO rule 11 in turn implies that any subsequent store operations will appear later in the global memory order than the store operation generated by the SC. However, since control, address, and data dependencies are defined over memory operations, and since an unsuccessful SC does not generate a memory operation, no order is enforced between unsuccessful SC and its dependent instructions. Moreover, since SC is defined to carry dependencies from its source registers to *rd* only when the SC is successful, an unsuccessful SC has no effect on the global memory order.

Table 82. A variant of the LB litmus test (outcome forbidden)

Initial values: 0(s0)=1; 0(s2)=1			
Hart 0		Hart 1	
(a)	ld a0,0(s0)	(e)	ld a3,0(s2)
(b)	lr a1,0(s1)	(f)	sd a3,0(s0)
(c)	sc a2,a0,0(s1)		
(d)	sd a2,0(s2)		
Outcome: `a0=0`, `a3=0`			



In addition, the choice to respect dependencies originating at store-conditional instructions ensures that certain out-of-thin-air-like behaviors will be prevented. Consider Table 82. Suppose a hypothetical implementation could occasionally make some early guarantee that a store-conditional operation will succeed. In this case, (c) could return 0 to `a2` early (before actually executing), allowing the sequence (d), (e), (f), (a), and then (b) to execute, and then (c) might execute (successfully) only at that point. This would imply that (c) writes its own success value to `0(s1)`! Fortunately, this situation and others like it are prevented by the fact that RVWMO respects dependencies originating at the stores generated by successful SC instructions.

We also note that syntactic dependencies between instructions only have any force when they take the form of a syntactic address, control, and/or data dependency. For example: a syntactic dependency between two `F` instructions via one of the `accumulating CSRs` in Section 18.3 does *not* imply that the two `F` instructions must be executed in order. Such a dependency would only serve to ultimately set up later a dependency from both `F` instructions to a later CSR instruction accessing the CSR flag in question.

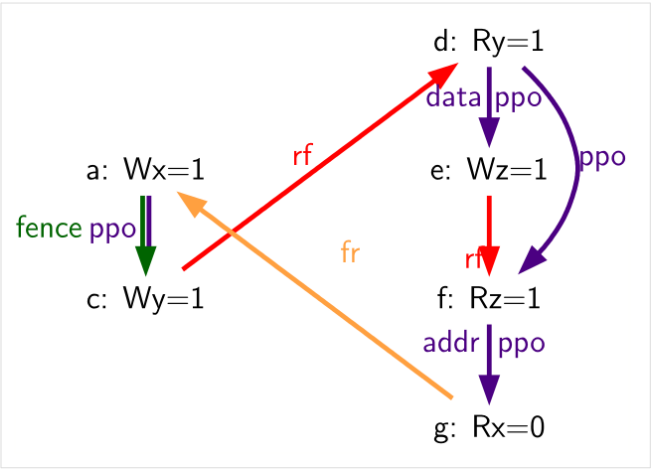
A.3.9. Pipeline Dependencies (Rules 12-13)

Rule 12: b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m

Rule 13: b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a

Table 83. Because of PPO rule 12 and the data dependency from (d) to (e), (d) must also precede (f) in the global memory order (outcome forbidden)

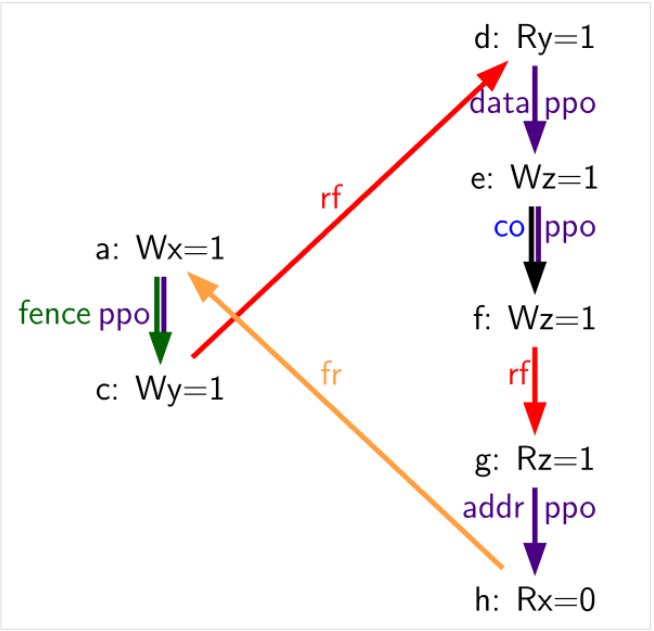
Hart 0		Hart 1	
	li t1, 1	(d)	lw a0, 0(s1)
(a)	sw t1,0(s0)	(e)	sw a0, 0(s2)
(b)	fence w, w	(f)	lw a1, 0(s2)
(c)	sw t1,0(s1)		xor a2,a1,a1
			add s0,s0,a2
		(g)	lw a3,0(s0)
Outcome: `a0=1`, `a3=0`			



PPO rules 12 and 13 reflect behaviors of almost all real processor pipeline implementations. Rule 12 states that a load cannot forward from a store until the address and data for that store are known. Consider Table 83 (f) cannot be executed until the data for (e) has been resolved, because (f) must return the value written by (e) (or by something even later in the global memory order), and the old value must not be clobbered by the write-back of (e) before (d) has had a chance to perform. Therefore, (f) will never perform before (d) has performed.

Table 84. Because of the extra store between (e) and (g), (d) no longer necessarily precedes (g) (outcome permitted)

Hart 0		Hart 1	
li t1, 1		li t1, 1	
(a)	sw t1,0(s0)	(d)	lw a0, 0(s1)
(b)	fence w, w	(e)	sw a0, 0(s2)
(c)	sw t1,0(s1)	(f)	sw t1, 0(s2)
		(g)	lw a1, 0(s2)
			xor a2,a1,a1
			add s0,s0,a2
		(h)	lw a3,0(s0)
Outcome: `a0=1`, `a3=0`			

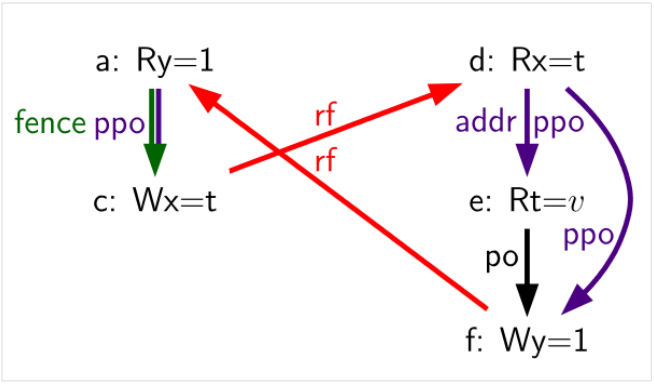


If there were another store to the same address in between (e) and (f), as in Table 85, then (f) would no longer be dependent on the data of (e) being resolved, and hence the dependency of (f) on (d), which produces the data for (e), would be broken.

Rule 13 makes a similar observation to the previous rule: a store cannot be performed at memory until all previous loads that might access the same address have themselves been performed. Such a load must appear to execute before the store, but it cannot do so if the store were to overwrite the value in memory before the load had a chance to read the old value. Likewise, a store generally cannot be performed until it is known that preceding instructions will not cause an exception due to failed address resolution, and in this sense, rule 13 can be seen as somewhat of a special case of rule 11.

Table 85. Because of the address dependency from (d) to (e), (d) also precedes (f) (outcome forbidden)

Hart 0		Hart 1	
		li t1, 1	
(a)	lw a0,0(s0)	(d)	lw a1, 0(s1)
(b)	fence rw,rw	(e)	lw a2, 0(a1)
(c)	sw s2,0(s1)	(f)	sw t1, 0(s0)
Outcome: `a0=1`, `a1=t`			



Consider Table 85 (f) cannot be executed until the address for (e) is resolved, because it may turn out that the addresses match; i.e., that `a1=s0`. Therefore, (f) cannot be sent to memory before (d) has executed and confirmed whether the addresses do indeed overlap.

A.4. Beyond Main Memory

RVWMO does not currently attempt to formally describe how FENCE.I, SFENCE.VMA, I/O fences, and PMAs behave. All of these behaviors will be described by future formalizations. In the meantime, the behavior of FENCE.I is described in Chapter 5, the behavior of SFENCE.VMA is described in the RISC-V Instruction Set Privileged Architecture Manual, and the behavior of I/O fences and the effects of PMAs are described below.

A.4.1. Coherence and Cacheability

The RISC-V Privileged ISA defines Physical Memory Attributes (PMAs) which specify, among other things, whether portions of the address space are coherent and/or cacheable. See the RISC-V Privileged ISA Specification for the complete details. Here, we simply discuss how the various details in each PMA relate to the memory model:

- Main memory vs.I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.
- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Cacheability PMAs: the cacheability PMAs in general do not affect the memory model. Non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless. However, some platform-specific and/or device-specific cacheability settings may differ.
- Coherence PMAs: The memory consistency model for memory regions marked as non-coherent in PMAs is currently platform-specific and/or device-specific: the load-value axiom, the atomicity axiom, and the progress axiom all may be violated with non-coherent memory. Note however that coherent memory does not require a hardware cache coherence protocol. The RISC-V Privileged ISA Specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise.
- Idempotency PMAs: Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

A.4.2. I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects and may return values other than the value "written" by the most recent store to the same address. Nevertheless, the following preserved program order rules still generally apply for accesses to I/O memory: memory access *a* precedes memory access *b* in global memory order if *a* precedes *b* in program order and one or more of the following holds:

1. *a* precedes *b* in preserved program order as defined in Chapter 18, with the exception that acquire and release ordering annotations apply only from one memory operation to another memory operation and from one I/O operation to another I/O operation, but not from a memory operation to an I/O nor vice versa
2. *a* and *b* are accesses to overlapping addresses in an I/O region
3. *a* and *b* are accesses to the same strongly ordered I/O region
4. *a* and *b* are accesses to I/O regions, and the channel associated with the I/O region accessed by either *a* or *b* is channel 1
5. *a* and *b* are accesses to I/O regions associated with the same channel (except for channel 0)

Note that the FENCE instruction distinguishes between main memory operations and I/O operations in its predecessor and successor sets. To enforce ordering between I/O operations and main memory operations, code must use a FENCE with PI, PO, SI, and/or SO, plus PR, PW, SR, and/or SW. For example, to enforce ordering between a write to main memory and an I/O write to a device register, a FENCE W,O or stronger is needed.

```
1 sd t0, 0(a0)
2 fence w,o
3 sd a0, 0(a1)
```

Listing 17. Ordering memory and I/O accesses

When a fence is in fact used, implementations must assume that the device may attempt to access memory immediately after receiving the MMIO signal, and subsequent memory accesses from that device to memory must observe the effects of all accesses ordered prior to that MMIO operation. In other words, in Listing 17, suppose `0(a0)` is in main memory and `0(a1)` is the address of a device register in I/O memory. If the device accesses `0(a0)` upon receiving the MMIO write, then that load must conceptually appear after the first store to `0(a0)` according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and main memory accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

Many architectures include separate notions of "ordering" and "completion" fences, especially as it relates to I/O (as opposed to regular main memory). Ordering fences simply ensure that memory operations stay in order, while completion fences ensure that predecessor accesses have all completed before any successors are made visible. RISC-V does not explicitly distinguish between ordering and completion fences. Instead, this distinction is simply inferred from different uses of the FENCE bits.

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices and DMA operations are required to access memory coherently and via strongly ordered I/O channels. Therefore, accesses to regular main memory regions that are concurrently accessed by external devices can also use the standard synchronization mechanisms. Implementations that do not conform to the Unix Platform

Specification and/or in which devices do not access memory coherently will need to use mechanisms (which are currently platform-specific or device-specific) to enforce coherency.

I/O regions in the address space should be considered non-cacheable regions in the PMAs for those regions. Such regions can be considered coherent by the PMA if they are not cached by any agent.

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe) may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

A.5. Code Porting and Mapping Guidelines

Table 86. Mappings from TSO operations to RISC-V operations

x86/TSO Operation	RVWMO Mapping
Load	`l{b|h|w|d}; fence r,rw`
Store	`fence rw,w; s{b|h|w|d}`
Atomic RMW	`amo<op>.{w|d}.aqrl OR` `loop:lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop`
Fence	`fence rw,rw`

Table 86 provides a mapping from TSO memory operations onto RISC-V memory instructions. Normal x86 loads and stores are all inherently acquire-RCpc and release-RCpc operations: TSO enforces all load-load, load-store, and store-store ordering by default. Therefore, under RVWMO, all TSO loads must be mapped onto a load followed by FENCE R,RW, and all TSO stores must be mapped onto FENCE RW,W followed by a store. TSO atomic read-modify-writes and x86 instructions using the LOCK prefix are fully ordered and can be implemented either via an AMO with both *aq* and *rl* set, or via an LR with *aq* set, the arithmetic operation in question, an SC with both *aq* and *rl* set, and a conditional branch checking the success condition. In the latter case, the *rl* annotation on the LR turns out (for non-obvious reasons) to be redundant and can be omitted.

Alternatives to Table 86 are also possible. A TSO store can be mapped onto AMOSWAP with *rl* set. However, since RVWMO PPO Rule 3 forbids forwarding of values from AMOs to subsequent loads, the use of AMOSWAP for stores may negatively affect performance. A TSO load can be mapped using LR with *aq* set: all such LR instructions will be unpaired, but that fact in and of itself does not preclude the use of LR for loads. However, again, this mapping may also negatively affect performance if it puts more pressure on the reservation mechanism than was originally intended.

Table 87. Mappings from Power operations to RISC-V operations

Power Operation	RVWMO Mapping
Load	`l{b|h|w|d}`
Load-Reserve	`lr.{w|d}`
Store	`s{b|h|w|d}`
Store-Conditional	`sc.{w|d}`
`lwsync`	`fence.tso`
`sync`	`fence rw,rw`
`isync`	`fence.i; fence r,r`

Table 87 provides a mapping from Power memory operations onto RISC-V memory instructions. Power

ISYNC maps on RISC-V to a FENCE.I followed by a FENCE R,R; the latter fence is needed because ISYNC is used to define a "control+control fence" dependency that is not present in RVWMO.

Table 88. Mappings from ARM operations to RISC-V operations

ARM Operation	RVWMO Mapping
Load	`l{b|h|w|d}`
Load-Acquire	`fence rw, rw; l{b|h|w|d}; fence r,rw`
Load-Exclusive	`lr.{w|d}`
Load-Acquire-Exclusive	`lr.{w|d}.aqrl`
Store	`s{b|h|w|d}`
Store-Release	`fence rw,w; s{b|h|w|d}`
Store-Exclusive	`sc.{w|d}`
Store-Release-Exclusive	`sc.{w|d}.rl`
`dmb`	`fence rw,rw`
`dmb.ld`	`fence r,rw`
`dmb.st`	`fence w,w`
`isb`	`fence.i; fence r,r`

Table 88 provides a mapping from ARM memory operations onto RISC-V memory instructions. Since RISC-V does not currently have plain load and store opcodes with *aq* or *rl* annotations, ARM load-acquire and store-release operations should be mapped using fences instead. Furthermore, in order to enforce store-release-to-load-acquire ordering, there must be a FENCE RW,RW between the store-release and load-acquire; Table 88 enforces this by always placing the fence in front of each acquire operation. ARM load-exclusive and store-exclusive instructions can likewise map onto their RISC-V LR and SC equivalents, but instead of placing a FENCE RW,RW in front of an LR with *aq* set, we simply also set *rl* instead. ARM ISB maps on RISC-V to FENCE.I followed by FENCE R,R similarly to how ISYNC maps for Power.

Table 89. Mappings from Linux memory primitives to RISC-V primitives.

Linux Operation	RVWMO Mapping
`smp_mb()`	`fence rw,rw`
`smp_rmb()`	`fence r,r`
`smp_wmb()`	`fence w,w`
`dma_rmb()`	`fence r,r`
`dma_wmb()`	`fence w,w`
`mb()`	`fence iorw,iorw`
`rmb()`	`fence ri,ri`
`wmb()`	`fence wo,wo`
`smp_load_acquire()`	`l{b|h|w|d}; fence r,rw`
`smp_store_release()`	`fence.tso; s{b|h|w|d}`
Linux Construct	RVWMO AMO Mapping
`atomic <op> relaxed`	`amo <op>.{w|d}`
`atomic <op> acquire`	`amo <op>.{w|d}.aq`
`atomic <op> release`	`amo <op>.{w|d}.rl`

Linux Operation	RVWMO Mapping
`atomic <op>`	`amo <op>.{w|d}.aqrl`
Linux Construct	RVWMO LR/SC Mapping
`atomic <op> relaxed`	`loop:lr.{w|d}; <op>; sc.{w|d}; bnez loop`
`atomic <op> acquire`	`loop:lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop`
`atomic <op> release`	`loop:lr.{w|d}; <op>; sc.{w|d}.aqrl*; bnez loop OR`
	`fence.tso; loop:lr.{w|d}; <op >; sc.{w|d}*; bnez loop`
`atomic <op>`	`loop:lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop`

With regards to Table 89, other constructs (such as spinlocks) should follow accordingly. Platforms or devices with non-coherent DMA may need additional synchronization (such as cache flush or invalidate mechanisms); currently any such extra synchronization will be device-specific.

Table 89 provides a mapping of Linux memory ordering macros onto RISC-V memory instructions. The Linux fences `dma_rmb()` and `dma_wmb()` map onto FENCE R,R and FENCE W,W, respectively, since the RISC-V Unix Platform requires coherent DMA, but would be mapped onto FENCE RI,RI and FENCE WO,WO, respectively, on a platform with non-coherent DMA. Platforms with non-coherent DMA may also require a mechanism by which cache lines can be flushed and/or invalidated. Such mechanisms will be device-specific and/or standardized in a future extension to the ISA.

The Linux mappings for release operations may seem stronger than necessary, but these mappings are needed to cover some cases in which Linux requires stronger orderings than the more intuitive mappings would provide. In particular, as of the time this text is being written, Linux is actively debating whether to require load-load, load-store, and store-store orderings between accesses in one critical section and accesses in a subsequent critical section in the same hart and protected by the same synchronization object. Not all combinations of FENCE RW,W/FENCE R,RW mappings with *aq*/*rl* mappings combine to provide such orderings. There are a few ways around this problem, including:

1. Always use FENCE RW,W/FENCE R,RW, and never use *aq*/*rl*. This suffices but is undesirable, as it defeats the purpose of the *aq*/*rl* modifiers.
2. Always use *aq*/*rl*, and never use FENCE RW,W/FENCE R,RW. This does not currently work due to the lack of load and store opcodes with *aq* and *rl* modifiers.
3. Strengthen the mappings of release operations such that they would enforce sufficient orderings in the presence of either type of acquire mapping. This is the currently recommended solution, and the one shown in Table 89.

RVWMO Mapping: (a) lw a0, 0(s0) (b) fence.tso // vs. fence rw,w (c) sd x0,0(s1) … loop: (d) amoswap.d.aq a1,t1,0(s1) bnez a1,loop (e) lw a2,0(s2)

For example, the critical section ordering rule currently being debated by the Linux community would require (a) to be ordered before (e) in Listing 18. If that will indeed be required, then it would be insufficient for (b) to map as FENCE RW,W. That said, these mappings are subject to change as the Linux Kernel Memory Model evolves.

```
Linux Code:
(a) int r0 = *x;
        (bc) spin_unlock(y, 0);
....
....
(d) spin_lock(y);
```

```
(e) int r1 = *z;

RVWMO Mapping:
(a) lw a0, 0(s0)
(b) fence.tso // vs. fence rw,w
(c) sd x0,0(s1)
....
loop:
(d) lr.d.aq a1,(s1)
bnez a1,loop
sc.d a1,t1,(s1)
bnez a1,loop
(e) lw a2,0(s2)
```

Listing 18. Orderings between critical sections in Linux

Table 90 provides a mapping of C11/C++11 atomic operations onto RISC-V memory instructions. If load and store opcodes with *aq* and *rl* modifiers are introduced, then the mappings in Table 91 will suffice. Note however that the two mappings only interoperate correctly if `atomic_<op>(memory_order_seq_cst)` is mapped using an LR that has both *aq* and *rl* set. Even more importantly, a Table 90 sequentially consistent store, followed by a Table 91 sequentially consistent load can be reordered unless the Table 90 mapping of stores is strengthened by either adding a second fence or mapping the store to `amoswap.rl` instead.

Table 90. Mappings from C/C++ primitives to RISC-V primitives.

C/C++ Construct	RVWMO Mapping
Non-atomic load	`l{b|h|w|d}`
`atomic_load(memory_order_relaxed)`	`l{b|h|w|d}`
`atomic_load(memory_order_acquire)`	`l{b|h|w|d}; fence r,rw`
`atomic_load(memory_order_seq_cst)`	`fence rw,rw; l{b|h|w|d}; fence r,rw`
Non-atomic store	`s{b|h|w|d}`
`atomic_store(memory_order_relaxed)`	`s{b|h|w|d}`
`atomic_store(memory_order_release)`	`fence rw,w; s{b|h|w|d}`
`atomic_store(memory_order_seq_cst)`	`fence rw,w; s{b|h|w|d}`
`atomic_thread_fence(memory_order_acquire)`	`fence r,rw`
`atomic_thread_fence(memory_order_release)`	`fence rw,w`
`atomic_thread_fence(memory_order_acq_rel)`	`fence.tso`
`atomic_thread_fence(memory_order_seq_cst)`	`fence rw,rw`
C/C++ Construct	RVWMO AMO Mapping
`atomic_<op>(memory_order_relaxed)`	`amo<op>.{w|d}`
`atomic_<op>(memory_order_acquire)`	`amo<op>.{w|d}.aq`
`atomic_<op>(memory_order_release)`	`amo<op>.{w|d}.rl`
`atomic_<op>(memory_order_acq_rel)`	`amo<op>.{w|d}.aqrl`
`atomic_<op>(memory_order_seq_cst)`	`amo<op>.{w|d}.aqrl`
C/C++ Construct	RVWMO LR/SC Mapping
`atomic_<op>(memory_order_relaxed)`	`loop:lr.{w|d}; <op>; sc.{w|d};`

C/C++ Construct	RVWMO Mapping
	`bnez loop`
`atomic_<op>(memory_order_acquire)`	`loop:lr.{w|d}.aq; <op>; sc.{w|d};`
	`bnez loop`
`atomic_<op>(memory_order_release)`	`loop:lr.{w|d}; <op>; sc.{w|d}.rl;`
	`bnez loop`
`atomic_<op>(memory_order_acq_rel)`	`loop:lr.{w|d}.aq; <op>; sc.{w|d}.rl;`
	`bnez loop`
`atomic_<op>(memory_order_seq_cst)`	`loop:lr.{w|d}.aqrl; <op>;`
	`sc.{w|d}.rl; bnez loop`

Table 91. Hypothetical mappings from C/C++ primitives to RISC-V primitives, if native load-acquire and store-release opcodes are introduced.

C/C++ Construct	RVWMO Mapping
Non-atomic load	`l{b|h|w|d}`
`atomic_load(memory_order_relaxed)`	`l{b|h|w|d}`
`atomic_load(memory_order_acquire)`	`l{b|h|w|d}.aq`
`atomic_load(memory_order_seq_cst)`	`l{b|h|w|d}.aq`
Non-atomic store	`s{b|h|w|d}`
`atomic_store(memory_order_relaxed)`	`s{b|h|w|d}`
`atomic_store(memory_order_release)`	`s{b|h|w|d}.rl`
`atomic_store(memory_order_seq_cst)`	`s{b|h|w|d}.rl`
`atomic_thread_fence(memory_order_acquire)`	`fence r,rw`
`atomic_thread_fence(memory_order_release)`	`fence rw,w`
`atomic_thread_fence(memory_order_acq_rel)`	`fence.tso`
`atomic_thread_fence(memory_order_seq_cst)`	`fence rw,rw`
C/C++ Construct	RVWMO AMO Mapping
`atomic_<op>(memory_order_relaxed)`	`amo<op>.{w|d}`
`atomic_<op>(memory_order_acquire)`	`amo<op>.{w|d}.aq`
`atomic_<op>(memory_order_release)`	`amo<op>.{w|d}.rl`
`atomic_<op>(memory_order_acq_rel)`	`amo<op>.{w|d}.aqrl`
`atomic_<op>(memory_order_seq_cst)`	`amo<op>.{w|d}.aqrl`
C/C++ Construct	RVWMO LR/SC Mapping
`atomic_<op>(memory_order_relaxed)`	`lr.{w|d}; <op>; sc.{w|d}`
`atomic_<op>(memory_order_acquire)`	`lr.{w|d}.aq; <op>; sc.{w|d}`
`atomic_<op>(memory_order_release)`	`lr.{w|d}; <op>; sc.{w|d}.rl`
`atomic_<op>(memory_order_acq_rel)`	`lr.{w|d}.aq; <op>; sc.{w|d}.rl`
`atomic_<op>(memory_order_seq_cst)`	`lr.{w|d}.aq`* `<op>; sc.{w|d}.rl`
* must be `lr.{w|d}.aqrl` in order to interoperate with code mapped per Table 90	

Any AMO can be emulated by an LR/SC pair, but care must be taken to ensure that any PPO orderings that originate from the LR are also made to originate from the SC, and that any PPO orderings that terminate at the SC are also made to terminate at the LR. For example, the LR must also be made to respect any data dependencies that the AMO has, given that load operations do not otherwise have any notion of a data dependency. Likewise, the effect a FENCE R,R elsewhere in the same hart must also be made to apply to the SC, which would not otherwise respect that fence. The emulator may achieve this effect by simply mapping AMOs onto `lr.aq; <op>; sc.aqrl`, matching the mapping used elsewhere for fully ordered atomics.

These C11/C++11 mappings require the platform to provide the following Physical Memory Attributes (as defined in the RISC-V Privileged ISA) for all memory:

- main memory
- coherent
- AMOArithmetic
- RsrvEventual

Platforms with different attributes may require different mappings, or require platform-specific SW (e.g., memory-mapped I/O).

A.6. Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design only admits executions that satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or *other-multi-copy-atomic*): any store value that is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no early inter-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer (unless of course it is done in such a way that no new illegal behaviors become architecturally visible). Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1 and 4-8 regularly and actively.

- expert programmers will use PPO rules 9-11 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 2-3 and 12-13 directly. These are included to facilitate common microarchitectural optimizations (rule 2) and the operational formal modeling approach (rules 3 and 12-13) described in Section B.3. They also facilitate the process of porting code from other architectures that have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1 and 3-6 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 2 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.
- PPO rule 7 may not be immediately obvious to architects, but it is a standard memory model requirement
- The load value axiom, the atomicity axiom, and PPO rules 8-13 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not `optimized away`.

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), regardless of the bits actually set
- implement all fences with PW and SR as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), as PW with SR is the most expensive of the four possible main memory ordering components anyway
- emulate *aq* and *rl* as described in Section A.5
- enforcing all same-address load-load ordering, even in the presence of patterns such as `fri-rfi` and `RSW`
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or LR to the same address
- forbid any forwarding of a value from an AMO or SC in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any main memory fences that do not include PW and SR ordering (e.g., as Ztso implementations will do)
- implement all atomics to be RCsc or even fully ordered, regardless of annotation

Architectures that implement RVTSO can safely do the following:

- Ignore all fences that do not have both PW and SR (unless the fence also orders I/O)
- Ignore all PPO rules except for rules 4 through 7, since the rest are redundant with other PPO rules under RVTSO assumptions

Other general notes:

- Silent stores (i.e., stores that write the same value that already exists at a memory location) behave like any other store from a memory model point of view. Likewise, AMOs which do not actually change the value in memory (e.g., an AMOMAX for which the value in *rs2* is smaller than the value currently in memory) are still semantically considered store operations. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in

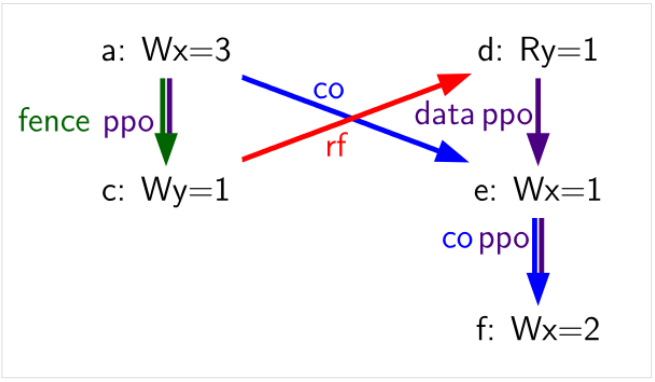
cases such as RSW Section A.3.5 which tend to be incompatible with silent stores.

- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

Table 92. Write subsumption litmus test, allowed execution

Hart 0		Hart 1	
li t1, 3		li t3, 2	
	li t2, 1		
(a)	sw t1,0(s0)	(d)	lw a0,0(s1)
(b)	fence w, w	(e)	sw a0,0(s0)
(c)	sw t2,0(s1)	(f)	sw t3,0(s0)



As written, if the load (d) reads value *1*, then (a) must precede (f) in the global memory order:

- (a) precedes (c) in the global memory order because of rule 4
- (c) precedes (d) in the global memory order because of the Load Value axiom
- (d) precedes (e) in the global memory order because of rule 10
- (e) precedes (f) in the global memory order because of rule 1

In other words the final value of the memory location whose address is in `s0` must be *2* (the value written by the store (f)) and cannot be *3* (the value written by the store (a)).

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

A.6.1. Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- "V" vector ISA extensions
- "J" JIT extension
- Native encodings for load and store opcodes with *aq* and *rl* set
- Fences limited to certain addresses
- Cache write-back/flush/invalidate/etc.instructions

A.7. Known Issues

A.7.1. Mixed-size RSW

Table 93. Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

Hart 0		Hart 1	
li t1, 1		li t1, 1	
(a)	lw a0,0(s0)	(d)	lw a1,0(s1)
(b)	fence rw,rw	(e)	amoswap.w.rl a2,t1,0(s2)
(c)	sw t1,0(s1)	(f)	ld a3,0(s2)
		(g)	lw a4,4(s2)
			xor a5,a4,a4
			add s0,s0,a5
		(h)	sw t1,0(s0)
Outcome: `a0=1`, `a1=1`, `a2=0`, `a3=1`, `a4=0`			

Table 94. Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

Hart 0		Hart 1	
li t1, 1		li t1, 1	
(a)	lw a0,0(s0)	(d)	ld a1,0(s1)
(b)	fence rw,rw	(e)	lw a2,4(s1)
(c)	sw t1,0(s1)		xor a3,a2,a2
			add s0,s0,a3
		(f)	sw t1,0(s0)
Outcome: `a0=1`, `a1=1`, `a2=0`			

Table 95. Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

Hart 0		Hart 1	
li t1, 1		li t1, 1	
(a)	lw a0,0(s0)	(d)	sw t1,4(s1)
(b)	fence rw,rw	(e)	ld a1,0(s1)
(c)	sw t1,0(s1)	(f)	lw a2,4(s1)
			xor a3,a2,a2
			add s0,s0,a3
		(g)	sw t1,0(s0)
Outcome: `a0=1`, `a1=0x100000001`, `a2=1`			

There is a known discrepancy between the operational and axiomatic specifications within the family of

mixed-size RSW variants shown in Table 93-Table 95. To address this, we may choose to add something like the following new PPO rule: Memory operation *a* precedes memory operation *b* in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program order, *a* and *b* both access regular main memory (rather than I/O regions), *a* is a load, *b* is a store, there is a load *m* between *a* and *b*, there is a byte *x* that both *a* and *m* read, there is no store between *a* and *m* that writes to *x*, and *m* precedes *b* in PPO. In other words, in herd syntax, we may choose to add `(po-loc & rsw);ppo;[W]` to PPO. Many implementations will already enforce this ordering naturally. As such, even though this rule is not official, we recommend that implementers enforce it nevertheless in order to ensure forwards compatibility with the possible future addition of this rule to RVWMO.

Appendix B: Formal Memory Model Specifications, Version 0.1

To facilitate formal analysis of RVWMO, this chapter presents a set of formalizations using different tools and modeling approaches. Any discrepancies are unintended; the expectation is that the models describe exactly the same sets of legal behaviors.

This appendix should be treated as commentary; all normative material is provided in Chapter 18 and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in Section A.7. Any other discrepancies are unintentional.

B.1. Formal Axiomatic Specification in Alloy

We present a formal specification of the RVWMO memory model in Alloy (alloy.mit.edu). This model is available online at github.com/daniellustig/riscv-memory-model.

The online material also contains some litmus tests and some examples of how Alloy can be used to model check some of the mappings in Section A.5.

```
// =RVWMO PPO=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store
  + rdw
  + (AMO + StoreConditional) <: rfi

  // explicit synchronization
  + ppo_fence
  + Acquire <: ^po :> MemoryEvent
  + MemoryEvent <: ^po :> Release
  + RCsc <: ^po :> RCsc
  + pair

  // syntactic dependencies
  + addrdep
  + datadep
  + ctrldep :> Store

  // pipeline dependencies
  + (addrdep+datadep).rfi
  + addrdep.^po :> Store
}

// the global memory order respects preserved program order
fact { ppo in ^gmo }
```

Listing 19. The RVWMO memory model formalized in Alloy (1/5: PPO)

```
// =RVWMO axioms=

// Load Value Axiom
fun candidates[r: MemoryEvent] : set MemoryEvent {
  (r.~^gmo & Store & same_addr[r]) // writes preceding r in gmo
  + (r.^~po & Store & same_addr[r]) // writes preceding r in po
}

fun latest_among[s: set Event] : Event { s - s.~^gmo }

pred LoadValue {
  all w: Store | all r: Load |
    w->r in rf <=> w = latest_among[candidates[r]]
}

// Atomicity Axiom
pred Atomicity {
  all r: Store.~pair |            // starting from the lr,
    no x: Store & same_addr[r] |  // there is no store x to the same addr
      x not in same_hart[r]       // such that x is from a different hart,
      and x in r.~rf.^gmo         // x follows (the store r reads from) in gmo,
      and r.pair in x.^gmo        // and r follows x in gmo
}

// Progress Axiom implicit: Alloy only considers finite executions

pred RISCV_mm { LoadValue and Atomicity /* and Progress */ }
```

The RVWMO memory model formalized in Alloy (2/5: Axioms)

```
//Basic model of memory

sig Hart {  // hardware thread
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // program order
}

abstract sig MemoryEvent extends Event {
  address: one Address,
  acquireRCpc: lone MemoryEvent,
  acquireRCsc: lone MemoryEvent,
  releaseRCpc: lone MemoryEvent,
  releaseRCsc: lone MemoryEvent,
  addrdep: set MemoryEvent,
  ctrldep: set Event,
  datadep: set MemoryEvent,
```

```
  gmo: set MemoryEvent,  // global memory order
  rf: set MemoryEvent
}
sig LoadNormal extends MemoryEvent {} // l{b|h|w|d}
sig LoadReserve extends MemoryEvent { // lr
  pair: lone StoreConditional
}
sig StoreNormal extends MemoryEvent {}       // s{b|h|w|d}
// all StoreConditionals in the model are assumed to be successful
sig StoreConditional extends MemoryEvent {}  // sc
sig AMO extends MemoryEvent {}               // amo
sig NOP extends Event {}

fun Load : Event { LoadNormal + LoadReserve + AMO }
fun Store : Event { StoreNormal + StoreConditional + AMO }

sig Fence extends Event {
  pr: lone Fence, // opcode bit
  pw: lone Fence, // opcode bit
  sr: lone Fence, // opcode bit
  sw: lone Fence  // opcode bit
}
sig FenceTSO extends Fence {}

/* Alloy encoding detail: opcode bits are either set (encoded, e.g.,
 * as f.pr in iden) or unset (f.pr not in iden).  The bits cannot be used for
 * anything else */
fact { pr + pw + sr + sw in iden }
// likewise for ordering annotations
fact { acquireRCpc + acquireRCsc + releaseRCpc + releaseRCsc in iden }
// don't try to encode FenceTSO via pr/pw/sr/sw; just use it as-is
fact { no FenceTSO.(pr + pw + sr + sw) }
```

Listing 20. The RVWMO memory model formalized in Alloy (3/5: model of memory)

```
// =Basic model rules=

// Ordering annotation groups
fun Acquire : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.acquireRCsc }
fun Release : MemoryEvent { MemoryEvent.releaseRCpc + MemoryEvent.releaseRCsc }
fun RCpc : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.releaseRCpc }
fun RCsc : MemoryEvent { MemoryEvent.acquireRCsc + MemoryEvent.releaseRCsc }

// There is no such thing as store-acquire or load-release, unless it's both
fact { Load & Release in Acquire }
fact { Store & Acquire in Release }

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
```

```
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
    (Load  <: ^po :> FencePRSR).(^po :> Load)
  + (Load  <: ^po :> FencePRSW).(^po :> Store)
  + (Store <: ^po :> FencePWSR).(^po :> Load)
  + (Store <: ^po :> FencePWSW).(^po :> Store)
  + (Load  <: ^po :> FenceTSO) .(^po :> MemoryEvent)
  + (Store <: ^po :> FenceTSO) .(^po :> Store)
}

// auxiliary definitions
fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// initial stores
fun NonInit : set Event { Hart.start.*po }
fun Init : set Event { Event - NonInit }
fact { Init in StoreNormal }
fact { Init->(MemoryEvent & NonInit) in ^gmo }
fact { all e: NonInit | one e.*~po.~start }  // each event is in exactly one
hart
fact { all a: Address | one Init & a.~address } // one init store per address
fact { no Init <: po and no po :> Init }
```

Listing 21. The RVWMO memory model formalized in Alloy (4/5: Basic model rules)

```
// po
fact { acyclic[po] }

// gmo
fact { total[^gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf.~rf in iden } // each read returns the value of only one write
fact { rf in Store <: address.~address :> Load }
fun rfi : MemoryEvent->MemoryEvent { rf & (*po + *~po) }

//dep
fact { no StoreNormal <: (addrdep + ctrldep + datadep) }
fact { addrdep + ctrldep + datadep + pair in ^po }
fact { datadep in datadep :> Store }
fact { ctrldep.*po in ctrldep }
fact { no pair & (^po :> (LoadReserve + StoreConditional)).^po }
fact { StoreConditional in LoadReserve.pair } // assume all SCs succeed

// rdw
```

```
fun rdw : Event->Event {
  (Load <: po_loc :> Load)  // start with all same_address load-load pairs,
  - (~rf.rf)                // subtract pairs that read from the same store,
  - (po_loc.rfi)            // and subtract out "fri-rfi" patterns
}

// filter out redundant instances and/or visualizations
fact { no gmo & gmo.gmo } // keep the visualization uncluttered
fact { all a: Address | some a.~address }

// =Optional: opcode encoding restrictions=

// the list of blessed fences
fact { Fence in
  Fence.pr.sr
  + Fence.pw.sw
  + Fence.pr.pw.sw
  + Fence.pr.sr.sw
  + FenceTSO
  + Fence.pr.pw.sr.sw
}

pred restrict_to_current_encodings {
  no (LoadNormal + StoreNormal) & (Acquire + Release)
}

// =Alloy shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
  all disj e, f: bag | e->f in rel + ~rel
  acyclic[rel]
}
```

Listing 22. The RVWMO memory model formalized in Alloy (5/5: Auxiliaries)

B.2. Formal Axiomatic Specification in Herd

The tool herd takes a memory model and a litmus test as input and simulates the execution of the test on top of the memory model. Memory models are written in the domain specific language Cat. This section provides two Cat memory model of RVWMO. The first model, Listing 24, follows the *global memory order*, Chapter 18, definition of RVWMO, as much as is possible for a Cat model. The second model, Listing 25, is an equivalent, more efficient, partial order based RVWMO model.

The simulator `herd` is part of the `diy` tool suite — see diy.inria.fr for software and documentation. The models and more are available online at diy.inria.fr/cats7/riscv/.

```
(*************)
(* Utilities *)
(*************)
```

```
(* All fence relations *)
let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]
let fence.tso =
  let f = fencerel(Fence.tso) in
  ([W];f;[W]) | ([R];f;[M])

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw |
  fence.tso

(* Same address, no W to the same address in-between *)
let po-loc-no-w = po-loc \ (po-loc?;[W];po-loc)
(* Read same write *)
let rsw = rf^-1;rf
(* Acquire, or stronger  *)
let AQ = Acq|AcqRel
(* Release or stronger *)
and RL = RelAcqRel
(* All RCsc *)
let RCsc = Acq|Rel|AcqRel
(* Amo events are both R and W, relation rmw relates paired lr/sc *)
let AMO = R & W
let StCond = range(rmw)

(*************)
(* ppo rules *)
(*************)

(* Overlapping-Address Orderings *)
let r1 = [M];po-loc;[W]
and r2 = ([R];po-loc-no-w;[R]) \ rsw
and r3 = [AMO|StCond];rfi;[R]
(* Explicit Synchronization *)
and r4 = fence
and r5 = [AQ];po;[M]
and r6 = [M];po;[RL]
and r7 = [RCsc];po;[RCsc]
and r8 = rmw
(* Syntactic Dependencies *)
and r9 = [M];addr;[M]
```

```
and r10 = [M];data;[W]
and r11 = [M];ctrl;[W]
(* Pipeline Dependencies *)
and r12 = [R];(addr|data);[W];rfi;[R]
and r13 = [R];addr;[M];po;[W]

let ppo = r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13
```

Listing 23. riscv-defs.cat, a herd definition of preserved program order (1/3)

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(*********************************)
(* Generate global memory order *)
(*********************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write after any write to the same location
  ppo |               # ppo compatible
  rfe                 # includes herd external rf (optimization)

(* Walk over all linear extensions of gmo0 *)
with  gmo from linearizations(M\IW,gmo0)

(* Add initial writes upfront -- convenient for computing rfGMO *)
let gmo = gmo | loc & IW * (M\IW)

(**********)
(* Axioms *)
(**********)

(* Compute rf according to the load value axiom, aka rfGMO *)
let WR = loc & ([W];(gmo|po);[R])
let rfGMO = WR \ (loc&([W];gmo);WR)

(* Check equality of herd rf and of rfGMO *)
empty (rf\rfGMO)|(rfGMO\rf) as RfCons

(* Atomicity axiom *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext
let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

Listing 24. riscv.cat, a herd version of the RVWMO memory model (2/3)

```
Partial

(***************)
(* Definitions *)
(***************)

(* Define ppo *)
include "riscv-defs.cat"

(* Compute coherence relation *)
include "cos-opt.cat"

(**********)
(* Axioms *)
(**********)

(* Sc per location *)
acyclic co|rf|fr|po-loc as Coherence

(* Main model axiom *)
acyclic co|rfe|fr|ppo as Model

(* Atomicity axiom *)
empty rmw & (fre;coe) as Atomic
```

Listing 25. `riscv.cat`*, an alternative herd presentation of the RVWMO memory model (3/3)*

B.3. An Operational Memory Model

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behavior as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavor: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from more implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behavior of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

The operational model, together with a fragment of the RISC-V ISA semantics (RV64I and A), are integrated into the `rmem` exploration tool (github.com/rems-project/rmem). `rmem` can explore litmus tests (see Section A.2) and small ELF binaries exhaustively, pseudorandomly and interactively. In `rmem`, the ISA semantics is expressed explicitly in Sail (see github.com/rems-project/sail for the Sail language, and github.com/rems-project/sail-riscv for the RISC-V ISA model), and the concurrency semantics is expressed in Lem (see github.com/rems-project/lem for the Lem language).

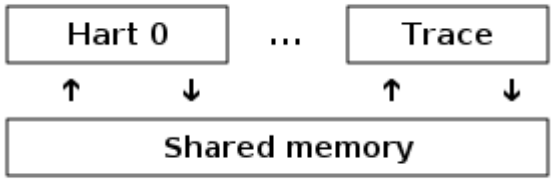
`rmem` has a command-line interface and a web-interface. The web-interface runs entirely on the client side, and is provided online together with a library of litmus tests: www.cl.cam.ac.uk/. The command-line interface is faster than the web-interface, specially in exhaustive mode.

Below is an informal introduction of the model states and transitions. The description of the formal model starts in the next subsection.

Terminology: In contrast to the axiomatic presentation, here every memory operation is either a load or a store. Hence, AMOs give rise to two distinct memory operations, a load and a store. When used in conjunction with `instruction`, the terms `load` and `store` refer to instructions that give rise to such memory operations. As such, both include AMO instructions. The term `acquire` refers to an instruction (or its memory operation) with the acquire-RCpc or acquire-RCsc annotation. The term `release` refers to an instruction (or its memory operation) with the release-RCpc or release-RCsc annotation.

Model states

Model states: A model state consists of a shared memory and a tuple of hart states.



The shared memory state records all the memory store operations that have propagated so far, in the order they propagated (this can be made more efficient, but for simplicity of the presentation we keep it this way).

Each hart state consists principally of a tree of instruction instances, some of which have been *finished*, and some of which have not. Non-finished instruction instances can be subject to *restart*, e.g. if they depend on an out-of-order or speculative load that turns out to be unsound.

Conditional branch and indirect jump instructions may have multiple successors in the instruction tree. When such instruction is finished, any untaken alternative paths are discarded.

Each instruction instance in the instruction tree has a state that includes an execution state of the intra-instruction semantics (the ISA pseudocode for this instruction). The model uses a formalization of the intra-instruction semantics in Sail. One can think of the execution state of an instruction as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information about the instance's memory and register footprints, its register reads and writes, its memory operations, whether it is finished, etc.

Model transitions

The model defines, for any model state, the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Execution of a single instruction will typically involve many transitions, and they may be interleaved in operational-model execution with transitions arising from other instructions. Each transition arises from a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its hart state and the shared memory state, but it does not depend on other hart states, and it will not change them. The transitions are introduced below and defined in Section B.3.5, with a precondition and a construction of the post-transition model state for each.

Transitions for all instructions:

- Fetch instruction: This transition represents a fetch and decode of a new instruction instance, as a program order successor of a previously fetched instruction instance (or the initial fetch address).

The model assumes the instruction memory is fixed; it does not describe the behavior of self-modifying code. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.

- Register write: This is a write of a register value.
- Register read: This is a read of a register value from the most recent program-order-predecessor instruction instance that writes to that register.
- Pseudocode internal step: This covers pseudocode internal computation: arithmetic, function calls, etc.
- Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted, memory accesses cannot be discarded, and all memory effects have taken place. For conditional branch and indirect jump instructions, any program order successors that were fetched from an address that is not the one that was written to the *pc* register are discarded, together with the sub-tree of instruction instances below them.

Transitions specific to load instructions:

- Initiate memory load operations: At this point the memory footprint of the load instruction is provisionally known (it could change if earlier instructions are restarted) and its individual memory load operations can start being satisfied.
- Satisfy memory load operation by forwarding from unpropogated stores: This partially or entirely satisfies a single memory load operation by forwarding, from program-order-previous memory store operations.
- Satisfy memory load operation from memory: This entirely satisfies the outstanding slices of a single memory load operation, from memory.
- Complete load operations: At this point all the memory load operations of the instruction have been entirely satisfied and the instruction pseudocode can continue executing. A load instruction can be subject to being restarted until the transition. But, under some conditions, the model might treat a load instruction as non-restartable even before it is finished (e.g. see).

Transitions specific to store instructions:

- Initiate memory store operation footprints: At this point the memory footprint of the store is provisionally known.
- Instantiate memory store operation values: At this point the memory store operations have their values and program-order-successor memory load operations can be satisfied by forwarding from them.
- Commit store instruction: At this point the store operations are guaranteed to happen (the instruction can no longer be restarted or discarded), and they can start being propagated to memory.
- Propagate store operation: This propagates a single memory store operation to memory.
- Complete store operations: At this point all the memory store operations of the instruction have been propagated to memory, and the instruction pseudocode can continue executing.

Transitions specific to `sc` instructions:

- Early sc fail: This causes the `sc` to fail, either a spontaneous fail or because it is not paired with a program-order-previous `lr`.
- Paired sc: This transition indicates the `sc` is paired with an `lr` and might succeed.
- Commit and propagate store operation of an sc: This is an atomic execution of the transitions Commit store instruction and Propagate store operation, it is enabled only if the stores from which the `lr` read from have not been overwritten.
- Late sc fail: This causes the `sc` to fail, either a spontaneous fail or because the stores from which the `lr` read from have been overwritten.

Transitions specific to AMO instructions:

- Satisfy, commit and propagate operations of an AMO: This is an atomic execution of all the transitions needed to satisfy the load operation, do the required arithmetic, and propagate the store operation.

Transitions specific to fence instructions:

- Commit fence

The transitions labeled ○ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behavior; the • cannot. Although Fetch instruction is marked with a •, it can be taken eagerly as long as it is not taken infinitely many times.

An instance of a non-AMO load instruction, after being fetched, will typically experience the following transitions in this order:

1. Register read
2. Initiate memory load operations
3. Satisfy memory load operation by forwarding from unpropagated stores and/or Satisfy memory load operation from memory (as many as needed to satisfy all the load operations of the instance)
4. Complete load operations
5. Register write
6. Finish instruction

Before, between, and after the transitions above, any number of Pseudocode internal step transitions may appear. In addition, a Fetch instruction transition for fetching the instruction in the next program location will be available until it is taken.

This concludes the informal description of the operational model. The following sections describe the formal operational model.

B.3.1. Intra-instruction Pseudocode Execution

The intra-instruction semantics for each instruction instance is expressed as a state machine, essentially running the instruction pseudocode. Given a pseudocode execution state, it computes the next state. Most states identify a pending memory or register operation, requested by the pseudocode, which the memory model has to do. The states are (this is a tagged union; tags in small-caps):

Load_mem(*kind*, *address*, *size*, *load_continuation*)	- memory load operation
Early_sc_fail(*res_continuation*)	- allow `sc` to fail early
Store_ea(*kind*, *address*, *size*, *next_state*)	- memory store effective address

Store_memv(*mem_value*, *store_continuation*)	- memory store value
Fence(*kind*, *next_state*)	- fence
Read_reg(*reg_name*, *read_continuation*)	- register read
Write_reg(*reg_name*, *reg_value*, *next_state*)	- register write
Internal(*next_state*)	- pseudocode internal step
Done	- end of pseudocode

Here:

- *mem_value* and *reg_value* are lists of bytes;
- *address* is an integer of XLEN bits;

for load/store, *kind* identifies whether it is `lr/sc`, acquire-RCpc/release-RCpc, acquire-RCsc/release-RCsc, acquire-release-RCsc; * for fence, *kind* identifies whether it is a normal or TSO, and (for normal fences) the predecessor and successor ordering bits; * *reg_name* identifies a register and a slice thereof (start and end bit indices); and the continuations describe how the instruction instance will continue for each value that might be provided by the surrounding memory model (the *load_continuation* and *read_continuation* take the value loaded from memory and read from the previous register write, the *store_continuation* takes *false* for an `sc` that failed and *true* in all other cases, and *res_continuation* takes *false* if the `sc` fails and *true* otherwise).

For example, given the load instruction `lw x1,0(x2)`*, an execution will typically go as follows. The initial execution state will be computed from the pseudocode for the given opcode. This can be expected to be Read_reg(*`x2`*, read_continuation). Feeding the most recently written value of register* `x2` *(the instruction semantics will be blocked if necessary until the register value is available), say* `0x4000`*, to read_continuation returns Load_mem(*`plain_load`*,* `0x4000`*,* `4`*, load_continuation). Feeding the 4-byte value loaded from memory location* `0x4000`*, say* `0x42`*, to load_continuation returns Write_reg(*`x1`*,* `0x42`*, Done). Many Internal(next_state) states may appear before and between the states above.*

Notice that writing to memory is split into two steps, Store_ea and Store_memv: the first one makes the memory footprint of the store provisionally known, and the second one adds the value to be stored. We ensure these are paired in the pseudocode (Store_ea followed by Store_memv), but there may be other steps between them.

It is observable that the Store_ea can occur before the value to be stored is determined. For example, for the litmus test LB+fence.r.rw+data-po to be allowed by the operational model (as it is by RVWMO), the first store in Hart 1 has to take the Store_ea step before its value is determined, so that the second store can see it is to a non-overlapping memory footprint, allowing the second store to be committed out of order without violating coherence.

The pseudocode of each instruction performs at most one store or one load, except for AMOs that perform exactly one load and one store. Those memory accesses are then split apart into the architecturally atomic units by the hart semantics (see Initiate memory load operations and Initiate memory store operation footprints below).

Informally, each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit (or from the hart's initial register state if there is no such write). Hence, it is essential to know the register write footprint of each instruction instance, which we calculate when the instruction instance is created (see the Fetch instruction action of below). We ensure in the pseudocode that each instruction does at most one register write to each register bit, and also that it does not try to read a register value it just wrote.

Data-flow dependencies (address and data) in the model emerge from the fact that each register read has to wait for the appropriate register write to be executed (as described above).

B.3.2. Instruction Instance State

Each instruction instance _*i* has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, AMO, fence, branch/jump or a `simple` instruction (this also includes a *kind* similar to the one described for the pseudocode execution states);
- *src_regs*, the set of source _reg_name_s (including system registers), as statically determined from the pseudocode of the instruction;
- *dst_regs*, the destination _reg_name_s (including system registers), as statically determined from the pseudocode of the instruction;
- *pseudocode_state* (or sometimes just `state` for short), one of (this is a tagged union; tags in small-caps):

Plain(*isa_state*)	**- ready to make a pseudocode transition**
Pending_mem_loads(*load_continuation*)	- requesting memory load operation(s)
Pending_mem_stores(*store_continuation*)	- requesting memory store operation(s)

- *reg_reads*, the register reads the instance has performed, including, for each one, the register write slices it read from;
- *reg_writes*, the register writes the instance has performed;
- *mem_loads*, a set of memory load operations, and for each one the as-yet-unsatisfied slices (the byte indices that have not been satisfied yet), and, for the satisfied slices, the store slices (each consisting of a memory store operation and subset of its byte indices) that satisfied it.
- *mem_stores*, a set of memory store operations, and for each one a flag that indicates whether it has been propagated (passed to the shared memory) or not.
- information recording whether the instance is committed, finished, etc.

Each memory load operation includes a memory footprint (address and size). Each memory store operations includes a memory footprint, and, when available, a value.

A load instruction instance with a non-empty *mem_loads*, for which all the load operations are satisfied (i.e. there are no unsatisfied load slices) is said to be *entirely satisfied*.

Informally, an instruction instance is said to have *fully determined data* if the load (and `sc`) instructions feeding its source registers are finished. Similarly, it is said to have a *fully determined memory footprint* if the load (and `sc`) instructions feeding its memory operation address register are finished. Formally, we first define the notion of *fully determined register write*: a register write *w* from *reg_writes* of instruction instance *i* is said to be *fully determined* if one of the following conditions hold:

1. *i* is finished; or
2. the value written by *w* is not affected by a memory operation that *i* has made (i.e. a value loaded from memory or the result of `sc`), and, for every register read that *i* has made, that affects *w*, the register write from which *i* read is fully determined (or *i* read from the initial register state).

Now, an instruction instance *i* is said to have *fully determined data* if for every register read *r* from *reg_reads*, the register writes that *r* reads from are fully determined. An instruction instance *i* is said to

have a *fully determined memory footprint* if for every register read *r* from *reg_reads* that feeds into *i*'s memory operation address, the register writes that *r* reads from are fully determined.

The `rmem` tool records, for every register write, the set of register writes from other instructions that have been read by this instruction at the point of performing the write. By carefully arranging the pseudocode of the instructions covered by the tool we were able to make it so that this is exactly the set of register writes on which the write depends on.

B.3.3. Hart State

The model state of a single hart comprises:

- *hart_id*, a unique identifier of the hart;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial instruction fetch address;
- *instruction_tree*, a tree of the instruction instances that have been fetched (and not discarded), in program order.

B.3.4. Shared Memory State

The model state of the shared memory comprises a list of memory store operations, in the order they propagated to the shared memory.

When a store operation is propagated to the shared memory it is simply added to the end of the list. When a load operation is satisfied from memory, for each byte of the load operation, the most recent corresponding store slice is returned.

For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to memory store operation slices, where each memory location is mapped to a one-byte slice of the most recent memory store operation to that location. However, this abstraction is not detailed enough to properly handle the `sc` instruction. The RVWMO allows store operations from the same hart as the `sc` to intervene between the store operation of the `sc` and the store operations the paired `lr` read from. To allow such store operations to intervene, and forbid others, the array abstraction must be extended to record more information. Here, we use a list as it is very simple, but a more efficient and scalable implementations should probably use something better.

B.3.5. Transitions

Each of the paragraphs below describes a single kind of system transition. The description starts with a condition over the current system state. The transition can be taken in the current state only if the condition is satisfied. The condition is followed by an action that is applied to that state when the transition is taken, in order to generate the new system state.

B.3.5.1. Fetch instruction

A possible program-order-successor of instruction instance *i* can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of *i* in the hart's *instruction_tree* are from *loc*; and
2. if *i*'s pseudocode has already written an address to *pc*, then *loc* must be that address, otherwise *loc* is:

- for a conditional branch, the successor address or the branch target address;
- for a (direct) jump and link instruction (`jal`), the target address;
- for an indirect jump instruction (`jalr`), any address; and
- for any other instruction, *i.program_loc*+4.

Action: construct a freshly initialized instruction instance *i'* for the instruction in the program memory at *loc*, with state Plain(*isa_state*), computed from the instruction pseudocode, including the static information available from the pseudocode such as its *instruction_kind*, *src_regs*, and *dst_regs*, and add *i'* to the hart's *instruction_tree* as a successor of *i*.

The possible next fetch addresses (*loc*) are available immediately after fetching *i* and the model does not need to wait for the pseudocode to write to *pc*; this allows out-of-order execution, and speculation past conditional branches and jumps. For most instructions these addresses are easily obtained from the instruction pseudocode. The only exception to that is the indirect jump instruction (`jalr`), where the address depends on the value held in a register. In principle the mathematical model should allow speculation to arbitrary addresses here. The exhaustive search in the `rmem` tool handles this by running the exhaustive search multiple times with a growing set of possible next fetch addresses for each indirect jump. The initial search uses empty sets, hence there is no fetch after indirect jump instruction until the pseudocode of the instruction writes to *pc*, and then we use that value for fetching the next instruction. Before starting the next iteration of exhaustive search, we collect for each indirect jump (grouped by code location) the set of values it wrote to *pc* in all the executions in the previous search iteration, and use that as possible next fetch addresses of the instruction. This process terminates when no new fetch addresses are detected.

B.3.5.2. Initiate memory load operations

An instruction instance *i* in state Plain(Load_mem(*kind*, *address*, *size*, *load_continuation*)) can always initiate the corresponding memory load operations. Action:

1. Construct the appropriate memory load operations *mlos*:
   - if *address* is aligned to *size* then *mlos* is a single memory load operation of *size* bytes from *address*;
   - otherwise, *mlos* is a set of *size* memory load operations, each of one byte, from the addresses *address*...*address*+*size*−1.
2. set *mem_loads* of *i* to *mlos*; and
3. update the state of *i* to Pending_mem_loads(*load_continuation*).

> In Section 18.1.1 it is said that misaligned memory accesses may be decomposed at any granularity. Here we decompose them to one-byte accesses as this granularity subsumes all others.

B.3.5.3. Satisfy memory load operation by forwarding from unpropagated stores

For a non-AMO load instruction instance *i* in state Pending_mem_loads(*load_continuation*), and a memory load operation *mlo* in *i.mem_loads* that has unsatisfied slices, the memory load operation can be partially or entirely satisfied by forwarding from unpropagated memory store operations by store instruction instances that are program-order-before *i* if:

1. all program-order-previous `fence` instructions with `.sr` and `.pw` set are finished;
2. for every program-order-previous `fence` instruction, *f*, with `.sr` and `.pr` set, and `.pw` not set, if *f* is not finished then all load instructions that are program-order-before *f* are entirely satisfied;

3. for every program-order-previous `fence.tso` instruction, *f*, that is not finished, all load instructions that are program-order-before *f* are entirely satisfied;
4. if *i* is a load-acquire-RCsc, all program-order-previous store-releases-RCsc are finished;
5. if *i* is a load-acquire-release, all program-order-previous instructions are finished;
6. all non-finished program-order-previous load-acquire instructions are entirely satisfied; and
7. all program-order-previous store-acquire-release instructions are finished;

Let *msoss* be the set of all unpropagated memory store operation slices from non-`sc` store instruction instances that are program-order-before *i* and have already calculated the value to be stored, that overlap with the unsatisfied slices of *mlo*, and which are not superseded by intervening store operations or store operations that are read from by an intervening load. The last condition requires, for each memory store operation slice *msos* in *msoss* from instruction *i'*:

- that there is no store instruction program-order-between *i* and *i'* with a memory store operation overlapping *msos*; and
- that there is no load instruction program-order-between *i* and *i'* that was satisfied from an overlapping memory store operation slice from a different hart.

Action:

1. update *i.mem_loads* to indicate that *mlo* was satisfied by *msoss*; and
2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction *i''* that is a program-order-successor of *i*, and every memory load operation *mlo''* of *i''* that was satisfied from *msoss'*, if there exists a memory store operation slice *msos'* in *msoss'*, and an overlapping memory store operation slice from a different memory store operation in *msoss*, and *msos'* is not from an instruction that is a program-order-successor of *i*, restart *i''* and its *restart-dependents*.

Where, the *restart-dependents* of instruction *j* are:

- program-order-successors of *j* that have data-flow dependency on a register write of *j*;
- program-order-successors of *j* that have a memory load operation that reads from a memory store operation of *j* (by forwarding);
- if *j* is a load-acquire, all the program-order-successors of *j*;
- if *j* is a load, for every `fence`, *f*, with `.sr` and `.pr` set, and `.pw` not set, that is a program-order-successor of *j*, all the load instructions that are program-order-successors of *f*;
- if *j* is a load, for every `fence.tso`, *f*, that is a program-order-successor of *j*, all the load instructions that are program-order-successors of *f*; and
- (recursively) all the restart-dependents of all the instruction instances above.

Forwarding memory store operations to a memory load might satisfy only some slices of the load, leaving other slices unsatisfied.

A program-order-previous store operation that was not available when taking the transition above might make *msoss* provisionally unsound (violating coherence) when it becomes available. That store will prevent the load from being finished (see Finish instruction), and will cause it to restart when that store operation is propagated (see Propagate store operation).

A consequence of the transition condition above is that store-release-RCsc memory store operations cannot be forwarded to load-acquire-RCsc instructions: *msoss* does not include memory store

operations from finished stores (as those must be propagated memory store operations), and the condition above requires all program-order-previous store-releases-RCsc to be finished when the load is acquire-RCsc.

B.3.5.4. Satisfy memory load operation from memory

For an instruction instance *i* of a non-AMO load instruction or an AMO instruction in the context of the Satisfy, commit and propagate operations of an AMO transition, any memory load operation *mlo* in *i.mem_loads* that has unsatisfied slices, can be satisfied from memory if all the conditions of <sat_by_forwarding, Satisfy memory load operation by forwarding from unpropagated stores>> are satisfied. Action: let *msoss* be the memory store operation slices from memory covering the unsatisfied slices of *mlo*, and apply the action of Satisfy memory operation by forwarding from unpropagates stores.

Note that Satisfy memory operation by forwarding from unpropagates stores might leave some slices of the memory load operation unsatisfied, those will have to be satisfied by taking the transition again, or taking Satisfy memory load operation from memory. Satisfy memory load operation from memory, on the other hand, will always satisfy all the unsatisfied slices of the memory load operation.

B.3.5.5. Complete load operations

A load instruction instance *i* in state Pending_mem_loads(*load_continuation*) can be completed (not to be confused with finished) if all the memory load operations *i.mem_loads* are entirely satisfied (i.e. there are no unsatisfied slices). Action: update the state of *i* to Plain(*load_continuation(mem_value)*), where *mem_value* is assembled from all the memory store operation slices that satisfied *i.mem_loads*.

B.3.5.6. Early sc fail

An `sc` instruction instance *i* in state Plain(Early_sc_fail(*res_continuation*)) can always be made to fail. Action: update the state of *i* to Plain(*res_continuation(false)*).

B.3.5.7. Paired sc

An `sc` instruction instance *i* in state Plain(Early_sc_fail(*res_continuation*)) can continue its (potentially successful) execution if *i* is paired with an `lr`. Action: update the state of *i* to Plain(*res_continuation(true)*).

B.3.5.8. Initiate memory store operation footprints

An instruction instance *i* in state Plain(Store_ea(*kind*, *address*, *size*, *next_state*)) can always announce its pending memory store operation footprint. Action:

1. construct the appropriate memory store operations *msos* (without the store value):
   - if *address* is aligned to *size* then *msos* is a single memory store operation of *size* bytes to *address*;
   - otherwise, *msos* is a set of *size* memory store operations, each of one-byte size, to the addresses *address*...*address*+*size*−1.
2. set *i.mem_stores* to *msos*; and
3. update the state of *i* to Plain(*next_state*).

Note that after taking the transition above the memory store operations do not yet have their values.

The importance of splitting this transition from the transition below is that it allows other program-order-successor store instructions to observe the memory footprint of this instruction, and if they don't overlap, propagate out of order as early as possible (i.e. before the data register value becomes available).

B.3.5.9. Instantiate memory store operation values

An instruction instance *i* in state Plain(Store_memv(*mem_value*, *store_continuation*)) can always instantiate the values of the memory store operations *i.mem_stores*. Action:

1. split *mem_value* between the memory store operations *i.mem_stores*; and
2. update the state of *i* to Pending_mem_stores(*store_continuation*).

B.3.5.10. Commit store instruction

An uncommitted instruction instance *i* of a non-`sc` store instruction or an `sc` instruction in the context of the Commit and propagate store operation of an `sc` transition, in state Pending_mem_stores(*store_continuation*), can be committed (not to be confused with propagated) if:

1. *i* has fully determined data;
2. all program-order-previous conditional branch and indirect jump instructions are finished;
3. all program-order-previous `fence` instructions with `.sw` set are finished;
4. all program-order-previous `fence.tso` instructions are finished;
5. all program-order-previous load-acquire instructions are finished;
6. all program-order-previous store-acquire-release instructions are finished;
7. if *i* is a store-release, all program-order-previous instructions are finished;
8. all program-order-previous memory access instructions have a fully determined memory footprint;
9. all program-order-previous store instructions, except for `sc` that failed, have initiated and so have non-empty *mem_stores*; and
10. all program-order-previous load instructions have initiated and so have non-empty *mem_loads*.

Action: record that *i* is committed.

Notice that if condition 8 is satisfied the conditions 9 and 10 are also satisfied, or will be satisfied after taking some eager transitions. Hence, requiring them does not strengthen the model. By requiring them, we guarantee that previous memory access instructions have taken enough transitions to make their memory operations visible for the condition check of , which is the next transition the instruction will take, making that condition simpler.

B.3.5.11. Propagate store operation

For a committed instruction instance *i* in state Pending_mem_stores(*store_continuation*), and an unpropagated memory store operation *mso* in *i.mem_stores*, *mso* can be propagated if:

1. all memory store operations of program-order-previous store instructions that overlap with *mso* have already propagated;
2. all memory load operations of program-order-previous load instructions that overlap with *mso* have already been satisfied, and (the load instructions) are *non-restartable* (see definition below); and

3. all memory load operations that were satisfied by forwarding *mso* are entirely satisfied.

Where a non-finished instruction instance *j* is *non-restartable* if:

1. there does not exist a store instruction *s* and an unpropagated memory store operation *mso* of *s* such that applying the action of the Propagate store operation transition to *mso* will result in the restart of *j*; and
2. there does not exist a non-finished load instruction *l* and a memory load operation *mlo* of *l* such that applying the action of the Satisfy memory load operation by forwarding from unpropagated stores /Satisfy memory load operation from memory transition (even if *mlo* is already satisfied) to *mlo* will result in the restart of *j*.

Action:

1. update the shared memory state with *mso*;
2. update *i.mem_stores* to indicate that *mso* was propagated; and
3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction *i'* program-order-after *i* and every memory load operation *mlo'* of *i'* that was satisfied from *msoss'*, if there exists a memory store operation slice *msos'* in *msoss'* that overlaps with *mso* and is not from *mso*, and *msos'* is not from a program-order-successor of *i*, restart *i'* and its *restart-dependents* (see Satisfy memory load operation by forwarding from unpropagated stores).

B.3.5.12. Commit and propagate store operation of an `sc`

An uncommitted `sc` instruction instance *i*, from hart *h*, in state Pending_mem_stores(*store_continuation*), with a paired `lr` *i'* that has been satisfied by some store slices *msoss*, can be committed and propagated at the same time if:

1. *i'* is finished;
2. every memory store operation that has been forwarded to *i'* is propagated;
3. the conditions of Commit store instruction is satisfied;
4. the conditions of Propagate store instruction is satisfied (notice that an `sc` instruction can only have one memory store operation); and
5. for every store slice *msos* from *msoss*, *msos* has not been overwritten, in the shared memory, by a store that is from a hart that is not *h*, at any point since *msos* was propagated to memory.

Action:

1. apply the actions of Commit store instruction; and
2. apply the action of Propagate store instruction.

B.3.5.13. Late `sc` fail

An `sc` instruction instance *i* in state Pending_mem_stores(*store_continuation*), that has not propagated its memory store operation, can always be made to fail. Action:

1. clear *i.mem_stores*; and
2. update the state of *i* to Plain(*store_continuation(false)*).

For efficiency, the `rmem` tool allows this transition only when it is not possible to take the Commit and propagate store operation of an sc transition. This does not affect the set of allowed final states, but when explored interactively, if the `sc` should fail one should use the Early sc fail transition instead of waiting for this transition.

B.3.5.14. Complete store operations

A store instruction instance *i* in state Pending_mem_stores(*store_continuation*), for which all the memory store operations in *i.mem_stores* have been propagated, can always be completed (not to be confused with finished). Action: update the state of *i* to Plain(*store_continuation(true)*).

B.3.5.15. Satisfy, commit and propagate operations of an AMO

An AMO instruction instance *i* in state Pending_mem_loads(*load_continuation*) can perform its memory access if it is possible to perform the following sequence of transitions with no intervening transitions:

1. Satisfy memory load operation from memory
2. Complere load operations
3. Pseudocode internal step (zero or more times)
4. Instantiate memory store operation values
5. Commit store instruction
6. Propagate store operation
7. Complete store operations

and in addition, the condition of Finish instruction, with the exception of not requiring *i* to be in state Plain(Done), holds after those transitions. Action: perform the above sequence of transitions (this does not include Finish instruction), one after the other, with no intervening transitions.

Notice that program-order-previous stores cannot be forwarded to the load of an AMO. This is simply because the sequence of transitions above does not include the forwarding transition. But even if it did include it, the sequence will fail when trying to do the Propagate store operation transition, as this transition requires all program-order-previous store operations to overlapping memory footprints to be propagated, and forwarding requires the store operation to be unpropagated.

In addition, the store of an AMO cannot be forwarded to a program-order-successor load. Before taking the transition above, the store operation of the AMO does not have its value and therefore cannot be forwarded; after taking the transition above the store operation is propagated and therefore cannot be forwarded.

B.3.5.16. Commit fence

A fence instruction instance *i* in state Plain(Fence(*kind*, *next_state*)) can be committed if:

1. if *i* is a normal fence and it has `.pr` set, all program-order-previous load instructions are finished;
2. if *i* is a normal fence and it has `.pw` set, all program-order-previous store instructions are finished; and
3. if *i* is a `fence.tso`, all program-order-previous load and store instructions are finished.

Action:

1. record that *i* is committed; and
2. update the state of *i* to Plain(*next_state*).

B.3.5.17. Register read

An instruction instance *i* in state Plain(Read_reg(*reg_name*, *read_cont*)) can do a register read of *reg_name* if every instruction instance that it needs to read from has already performed the expected *reg_name* register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *reg_value* be the value assembled from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *reg_value*; and
2. update the state of *i* to Plain(*read_cont*(*reg_value*)).

B.3.5.18. Register write

An instruction instance *i* in state Plain(Write_reg(*reg_name*, *reg_value*, *next_state*)) can always do a *reg_name* register write. Action:

1. add *reg_name* to *i.reg_writes* with *deps* and *reg_value*; and
2. update the state of *i* to Plain(*next_state*).

where *deps* is a pair of the set of all *read_sources* from *i.reg_reads*, and a flag that is true iff *i* is a load instruction instance that has already been entirely satisfied.

B.3.5.19. Pseudocode internal step

An instruction instance *i* in state Plain(Internal(*next_state*)) can always do that pseudocode-internal step. Action: update the state of *i* to Plain(*next_state*).

B.3.5.20. Finish instruction

A non-finished instruction instance *i* in state Plain(Done) can be finished if:

1. if *i* is a load instruction:
   a. all program-order-previous load-acquire instructions are finished;
   b. all program-order-previous `fence` instructions with `.sr` set are finished;
   c. for every program-order-previous `fence.tso` instruction, *f*, that is not finished, all load instructions that are program-order-before *f* are finished; and
   d. it is guaranteed that the values read by the memory load operations of *i* will not cause coherence violations, i.e., for any program-order-previous instruction instance *i'*, let *cfp* be the combined footprint of propagated memory store operations from store instructions program-order-between *i* and *i'*, and *fixed memory store operations* that were forwarded to *i* from store instructions program-order-between *i* and *i'* including *i'*, and let */cfp* be the complement of *cfp* in the memory footprint of *i*. If */cfp* is not empty:

i. *i'* has a fully determined memory footprint;

ii. *i'* has no unpropagated memory store operations that overlap with */cfp*; and

iii. if *i'* is a load with a memory footprint that overlaps with */cfp*, then all the memory load operations of *i'* that overlap with */cfp* are satisfied and *i'* is *non-restartable* (see the Propagate store operation transition for how to determined if an instruction is non-restartable).

Here, a memory store operation is called fixed if the store instruction has fully determined data.

2. *i* has a fully determined data; and
3. if *i* is not a fence, all program-order-previous conditional branch and indirect jump instructions are finished.

Action:

1. if *i* is a conditional branch or indirect jump instruction, discard any untaken paths of execution, i.e., remove all instruction instances that are not reachable by the branch/jump taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

B.3.6. Limitations

- The model covers user-level RV64I and RV64A. In particular, it does not support the misaligned atomicity granule PMA or the total store ordering extension "Ztso". It should be trivial to adapt the model to RV32I/A and to the G, Q and C extensions, but we have never tried it. This will involve, mostly, writing Sail code for the instructions, with minimal, if any, changes to the concurrency model.
- The model covers only normal memory accesses (it does not handle I/O accesses).
- The model does not cover TLB-related effects.
- The model assumes the instruction memory is fixed. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.
- The model does not cover exceptions, traps and interrupts.

Appendix C: Vector Assembly Code Examples

The following are provided as non-normative text to help explain the vector ISA.

C.1. Vector-vector add example

```
    # vector-vector add routine of 32-bit integers
    # void vvaddint32(size_t n, const int*x, const int*y, int*z)
    # { for (size_t i=0; i<n; i++) { z[i]=x[i]+y[i]; } }
    #
    # a0 = n, a1 = x, a2 = y, a3 = z
    # Non-vector instructions are indented
vvaddint32:
    vsetvli t0, a0, e32, m1, ta, ma  # Set vector length based on 32-bit
vectors
    vle32.v v0, (a1)         # Get first vector
      sub a0, a0, t0         # Decrement number done
      slli t0, t0, 2         # Multiply number done by 4 bytes
      add a1, a1, t0         # Bump pointer
    vle32.v v1, (a2)         # Get second vector
      add a2, a2, t0         # Bump pointer
    vadd.vv v2, v0, v1       # Sum vectors
    vse32.v v2, (a3)         # Store result
      add a3, a3, t0         # Bump pointer
      bnez a0, vvaddint32    # Loop back
      ret                    # Finished
```

C.2. Example with mixed-width mask and compute.

```
# Code using one width for predicate and different width for masked
# compute.
#   int8_t a[]; int32_t b[], c[];
#   for (i=0;  i<n; i++) { b[i] =  (a[i] < 5) ? c[i] : 1; }
#
# Mixed-width code that keeps SEW/LMUL=8
  loop:
    vsetvli a4, a0, e8, m1, ta, ma   # Byte vector for predicate calc
    vle8.v v1, (a1)               # Load a[i]
      add a1, a1, a4              # Bump pointer.
    vmslt.vi v0, v1, 5            # a[i] < 5?

    vsetvli x0, a0, e32, m4, ta, mu  # Vector of 32-bit values.
      sub a0, a0, a4              # Decrement count
    vmv.v.i v4, 1                 # Splat immediate to destination
    vle32.v v4, (a3), v0.t        # Load requested elements of C, others
undisturbed
      sll t1, a4, 2
```

```
      add a3, a3, t1             # Bump pointer.
    vse32.v v4, (a2)             # Store b[i].
      add a2, a2, t1             # Bump pointer.
      bnez a0, loop              # Any more?
```

C.3. Memcpy example

```
    # void *memcpy(void* dest, const void* src, size_t n)
    # a0=dest, a1=src, a2=n
    #
  memcpy:
      mv a3, a0 # Copy destination
  loop:
    vsetvli t0, a2, e8, m8, ta, ma   # Vectors of 8b
    vle8.v v0, (a1)                # Load bytes
      add a1, a1, t0               # Bump pointer
      sub a2, a2, t0               # Decrement count
    vse8.v v0, (a3)                # Store bytes
      add a3, a3, t0               # Bump pointer
      bnez a2, loop                # Any more?
      ret                          # Return
```

C.4. Conditional example

```
# (int16) z[i] = ((int8) x[i] < 5) ? (int16) a[i] : (int16) b[i];
#

loop:
    vsetvli t0, a0, e8, m1, ta, ma # Use 8b elements.
    vle8.v v0, (a1)          # Get x[i]
      sub a0, a0, t0         # Decrement element count
      add a1, a1, t0         # x[i] Bump pointer
    vmslt.vi v0, v0, 5       # Set mask in v0
    vsetvli x0, x0, e16, m2, ta, mu  # Use 16b elements.
      slli t0, t0, 1         # Multiply by 2 bytes
    vle16.v v2, (a2), v0.t   # z[i] = a[i] case
    vmnot.m v0, v0           # Invert v0
      add a2, a2, t0         # a[i] bump pointer
    vle16.v v2, (a3), v0.t   # z[i] = b[i] case
      add a3, a3, t0         # b[i] bump pointer
    vse16.v v2, (a4)         # Store z
      add a4, a4, t0         # z[i] bump pointer
      bnez a0, loop
```

C.5. SAXPY example

```
# void
# saxpy(size_t n, const float a, const float *x, float *y)
# {
#   size_t i;
#   for (i=0; i<n; i++)
#     y[i] = a * x[i] + y[i];
# }
#
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y

saxpy:
    vsetvli a4, a0, e32, m8, ta, ma
    vle32.v v0, (a1)
    sub a0, a0, a4
    slli a4, a4, 2
    add a1, a1, a4
    vle32.v v8, (a2)
    vfmacc.vf v8, fa0, v0
    vse32.v v8, (a2)
    add a2, a2, a4
    bnez a0, saxpy
    ret
```

C.6. SGEMM example

```
# RV64IDV system
#
# void
# sgemm_nn(size_t n,
#          size_t m,
#          size_t k,
#          const float*a,   // m * k matrix
#          size_t lda,
#          const float*b,   // k * n matrix
#          size_t ldb,
#          float*c,         // m * n matrix
#          size_t ldc)
#
#  c += a*b (alpha=1, no transpose on input matrices)
#  matrices stored in C row-major order
```

```
#define n a0
#define m a1
#define k a2
#define ap a3
#define astride a4
#define bp a5
#define bstride a6
#define cp a7
#define cstride t0
#define kt t1
#define nt t2
#define bnp t3
#define cnp t4
#define akp t5
#define bkp s0
#define nvl s1
#define ccp s2
#define amp s3

# Use args as additional temporaries
#define ft12 fa0
#define ft13 fa1
#define ft14 fa2
#define ft15 fa3

# This version holds a 16*VLMAX block of C matrix in vector registers
# in inner loop, but otherwise does not cache or TLB tiling.

sgemm_nn:
    addi sp, sp, -FRAMESIZE
    sd s0, OFFSET(sp)
    sd s1, OFFSET(sp)
    sd s2, OFFSET(sp)

    # Check for zero size matrices
    beqz n, exit
    beqz m, exit
    beqz k, exit

    # Convert elements strides to byte strides.
    ld cstride, OFFSET(sp)   # Get arg from stack frame
    slli astride, astride, 2
    slli bstride, bstride, 2
    slli cstride, cstride, 2

    slti t6, m, 16
    bnez t6, end_rows

c_row_loop: # Loop across rows of C blocks
```

```
    mv nt, n  # Initialize n counter for next row of C blocks

    mv bnp, bp # Initialize B n-loop pointer to start
    mv cnp, cp # Initialize C n-loop pointer

c_col_loop: # Loop across one row of C blocks
    vsetvli nvl, nt, e32, m1, ta, ma  # 32-bit vectors, LMUL=1

    mv akp, ap   # reset pointer into A to beginning
    mv bkp, bnp # step to next column in B matrix

    # Initialize current C submatrix block from memory.
    vle32.v  v0, (cnp); add ccp, cnp, cstride;
    vle32.v  v1, (ccp); add ccp, ccp, cstride;
    vle32.v  v2, (ccp); add ccp, ccp, cstride;
    vle32.v  v3, (ccp); add ccp, ccp, cstride;
    vle32.v  v4, (ccp); add ccp, ccp, cstride;
    vle32.v  v5, (ccp); add ccp, ccp, cstride;
    vle32.v  v6, (ccp); add ccp, ccp, cstride;
    vle32.v  v7, (ccp); add ccp, ccp, cstride;
    vle32.v  v8, (ccp); add ccp, ccp, cstride;
    vle32.v  v9, (ccp); add ccp, ccp, cstride;
    vle32.v v10, (ccp); add ccp, ccp, cstride;
    vle32.v v11, (ccp); add ccp, ccp, cstride;
    vle32.v v12, (ccp); add ccp, ccp, cstride;
    vle32.v v13, (ccp); add ccp, ccp, cstride;
    vle32.v v14, (ccp); add ccp, ccp, cstride;
    vle32.v v15, (ccp)


    mv kt, k # Initialize inner loop counter

    # Inner loop scheduled assuming 4-clock occupancy of vfmacc instruction and
single-issue pipeline
    # Software pipeline loads
    flw ft0, (akp); add amp, akp, astride;
    flw ft1, (amp); add amp, amp, astride;
    flw ft2, (amp); add amp, amp, astride;
    flw ft3, (amp); add amp, amp, astride;
    # Get vector from B matrix
    vle32.v v16, (bkp)

    # Loop on inner dimension for current C block
 k_loop:
    vfmacc.vf v0, ft0, v16
    add bkp, bkp, bstride
    flw ft4, (amp)
    add amp, amp, astride
    vfmacc.vf v1, ft1, v16
    addi kt, kt, -1    # Decrement k counter
```

```
    flw ft5, (amp)
    add amp, amp, astride
    vfmacc.vf v2, ft2, v16
    flw ft6, (amp)
    add amp, amp, astride
    flw ft7, (amp)
    vfmacc.vf v3, ft3, v16
    add amp, amp, astride
    flw ft8, (amp)
    add amp, amp, astride
    vfmacc.vf v4, ft4, v16
    flw ft9, (amp)
    add amp, amp, astride
    vfmacc.vf v5, ft5, v16
    flw ft10, (amp)
    add amp, amp, astride
    vfmacc.vf v6, ft6, v16
    flw ft11, (amp)
    add amp, amp, astride
    vfmacc.vf v7, ft7, v16
    flw ft12, (amp)
    add amp, amp, astride
    vfmacc.vf v8, ft8, v16
    flw ft13, (amp)
    add amp, amp, astride
    vfmacc.vf v9, ft9, v16
    flw ft14, (amp)
    add amp, amp, astride
    vfmacc.vf v10, ft10, v16
    flw ft15, (amp)
    add amp, amp, astride
    addi akp, akp, 4            # Move to next column of a
    vfmacc.vf v11, ft11, v16
    beqz kt, 1f                 # Don't load past end of matrix
    flw ft0, (akp)
    add amp, akp, astride
1:  vfmacc.vf v12, ft12, v16
    beqz kt, 1f
    flw ft1, (amp)
    add amp, amp, astride
1:  vfmacc.vf v13, ft13, v16
    beqz kt, 1f
    flw ft2, (amp)
    add amp, amp, astride
1:  vfmacc.vf v14, ft14, v16
    beqz kt, 1f                 # Exit out of loop
    flw ft3, (amp)
    add amp, amp, astride
    vfmacc.vf v15, ft15, v16
    vle32.v v16, (bkp)            # Get next vector from B matrix, overlap
```

```
loads with jump stalls
    j k_loop

1:  vfmacc.vf v15, ft15, v16

    # Save C matrix block back to memory
    vse32.v  v0, (cnp); add ccp, cnp, cstride;
    vse32.v  v1, (ccp); add ccp, ccp, cstride;
    vse32.v  v2, (ccp); add ccp, ccp, cstride;
    vse32.v  v3, (ccp); add ccp, ccp, cstride;
    vse32.v  v4, (ccp); add ccp, ccp, cstride;
    vse32.v  v5, (ccp); add ccp, ccp, cstride;
    vse32.v  v6, (ccp); add ccp, ccp, cstride;
    vse32.v  v7, (ccp); add ccp, ccp, cstride;
    vse32.v  v8, (ccp); add ccp, ccp, cstride;
    vse32.v  v9, (ccp); add ccp, ccp, cstride;
    vse32.v v10, (ccp); add ccp, ccp, cstride;
    vse32.v v11, (ccp); add ccp, ccp, cstride;
    vse32.v v12, (ccp); add ccp, ccp, cstride;
    vse32.v v13, (ccp); add ccp, ccp, cstride;
    vse32.v v14, (ccp); add ccp, ccp, cstride;
    vse32.v v15, (ccp)

    # Following tail instructions should be scheduled earlier in free slots
during C block save.
    # Leaving here for clarity.

    # Bump pointers for loop across blocks in one row
    slli t6, nvl, 2
    add cnp, cnp, t6                        # Move C block pointer over
    add bnp, bnp, t6                        # Move B block pointer over
    sub nt, nt, nvl                         # Decrement element count in n
dimension
    bnez nt, c_col_loop                     # Any more to do?

    # Move to next set of rows
    addi m, m, -16  # Did 16 rows above
    slli t6, astride, 4  # Multiply astride by 16
    add ap, ap, t6         # Move A matrix pointer down 16 rows
    slli t6, cstride, 4  # Multiply cstride by 16
    add cp, cp, t6         # Move C matrix pointer down 16 rows

    slti t6, m, 16
    beqz t6, c_row_loop

    # Handle end of matrix with fewer than 16 rows.
    # Can use smaller versions of above decreasing in powers-of-2 depending on
code-size concerns.
end_rows:
    # Not done.
```

```
exit:
    ld s0, OFFSET(sp)
    ld s1, OFFSET(sp)
    ld s2, OFFSET(sp)
    addi sp, sp, FRAMESIZE
    ret
```

C.7. Division approximation example

```
# v1 = v1 / v2 to almost 23 bits of precision.

vfrec7.v v3, v2             # Estimate 1/v2
  li t0, 0x3f800000
vmv.v.x v4, t0              # Splat 1.0
vfnmsac.vv v4, v2, v3       # 1.0 - v2 * est(1/v2)
vfmadd.vv v3, v4, v3        # Better estimate of 1/v2
vmv.v.x v4, t0              # Splat 1.0
vfnmsac.vv v4, v2, v3       # 1.0 - v2 * est(1/v2)
vfmadd.vv v3, v4, v3        # Better estimate of 1/v2
vfmul.vv v1, v1, v3         # Estimate of v1/v2
```

C.8. Square root approximation example

```
# v1 = sqrt(v1) to more than 23 bits of precision.

  fmv.w.x ft0, x0           # Mask off zero inputs
vmfne.vf v0, v1, ft0        #   to avoid DZ exception
vfrsqrt7.v v2, v1, v0.t     # Estimate r ~= 1/sqrt(v1)
vmfne.vf v0, v2, ft0, v0.t  # Mask off +inf to avoid NV
  li t0, 0x3f800000
  fli.s ft0, 0.5
vmv.v.x v5, t0              # Splat 1.0
vfmul.vv v3, v1, v2, v0.t   # t = v1 r
vfmul.vf v4, v2, ft0, v0.t  # 0.5 r
vfmsub.vv v3, v2, v5, v0.t  # t r - 1
vfnmsac.vv v2, v3, v4, v0.t # r - (0.5 r) (t r - 1)
                            # Better estimate of 1/sqrt(v1)
vfmul.vv v1, v1, v2, v0.t   # t = v1 r
vfmsub.vv v2, v1, v5, v0.t  # t r - 1
vfmul.vf v3, v1, ft0, v0.t  # 0.5 t
vfnmsac.vv v1, v2, v3, v0.t # t - (0.5 t) (t r - 1)
                            # ~ sqrt(v1) to about 23.3 bits
```

C.9. C standard library strcmp example

```
  # int strcmp(const char *src1, const char* src2)
strcmp:
    ##  Using LMUL=2, but same register names work for larger LMULs
    li t1, 0                # Initial pointer bump
loop:
    vsetvli t0, x0, e8, m2, ta, ma  # Max length vectors of bytes
    add a0, a0, t1          # Bump src1 pointer
    vle8ff.v v8, (a0)       # Get src1 bytes
    add a1, a1, t1          # Bump src2 pointer
    vle8ff.v v16, (a1)      # Get src2 bytes

    vmseq.vi v0, v8, 0      # Flag zero bytes in src1
    vmsne.vv v1, v8, v16    # Flag if src1 != src2
    vmor.mm v0, v0, v1      # Combine exit conditions

    vfirst.m a2, v0         # ==0 or != ?
    csrr t1, vl             # Get number of bytes fetched

    bltz a2, loop           # Loop if all same and no zero byte

    add a0, a0, a2          # Get src1 element address
    lbu a3, (a0)            # Get src1 byte from memory

    add a1, a1, a2          # Get src2 element address
    lbu a4, (a1)            # Get src2 byte from memory

    sub a0, a3, a4          # Return value.

    ret
```

C.10. Fractional Lmul example

This appendix presents a non-normative example to help explain where compilers can make good use of the fractional LMUL feature.

Consider the following (admittedly contrived) loop written in C:

```
void add_ref(long N,
    signed char *restrict c_c, signed char *restrict c_a, signed char *restrict
c_b,
    long *restrict l_c, long *restrict l_a, long *restrict l_b,
    long *restrict l_d, long *restrict l_e, long *restrict l_f,
    long *restrict l_g, long *restrict l_h, long *restrict l_i,
    long *restrict l_j, long *restrict l_k, long *restrict l_l,
    long *restrict l_m) {
  long i;
```

```
  for (i = 0; i < N; i++) {
    c_c[i] = c_a[i] + c_b[i]; // Note this 'char' addition that creates a mixed
type situation
    l_c[i] = l_a[i] + l_b[i];
    l_f[i] = l_d[i] + l_e[i];
    l_i[i] = l_g[i] + l_h[i];
    l_l[i] = l_k[i] + l_j[i];
    l_m[i] += l_m[i] + l_c[i] + l_f[i] + l_i[i] + l_l[i];
  }
}
```

The example loop has a high register pressure due to the many input variables and temporaries required. The compiler realizes there are two datatypes within the loop: an 8-bit 'char' and a 64-bit 'long *'. Without fractional LMUL, the compiler would be forced to use LMUL=1 for the 8-bit computation and LMUL=8 for the 64-bit computation(s), to have equal number of elements on all computations within the same loop iteration. Under LMUL=8, only 4 registers are available to the register allocator. Given the large number of 64-bit variables and temporaries required in this loop, the compiler ends up generating a lot of spill code. The code below demonstrates this effect:

```
.LBB0_4:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
    add     s9, a2, s6
    vsetvli s1, zero, e8,m1,ta,mu
    vle8.v  v25, (s9)
    add     s1, a3, s6
    vle8.v  v26, (s1)
    vadd.vv v25, v26, v25
    add     s1, a1, s6
    vse8.v  v25, (s1)
    add     s9, a5, s10
    vsetvli s1, zero, e64,m8,ta,mu
    vle64.v v8, (s9)
    add s1, a6, s10
    vle64.v v16, (s1)
    add     s1, a7, s10
    vle64.v v24, (s1)
    add     s1, s3, s10
    vle64.v v0, (s1)
    sd      a0, -112(s0)
    ld      a0, -128(s0)
    vs8r.v  v0, (a0) # Spill LMUL=8
    add     s9, t6, s10
    add     s11, t5, s10
    add     ra, t2, s10
    add     s1, t3, s10
    vle64.v v0, (s9)
    ld      s9, -136(s0)
    vs8r.v  v0, (s9) # Spill LMUL=8
    vle64.v v0, (s11)
```

```
    ld      s9, -144(s0)
    vs8r.v  v0, (s9) # Spill LMUL=8
    vle64.v v0, (ra)
    ld      s9, -160(s0)
    vs8r.v  v0, (s9) # Spill LMUL=8
    vle64.v v0, (s1)
    ld      s1, -152(s0)
    vs8r.v  v0, (s1) # Spill LMUL=8
    vadd.vv v16, v16, v8
    ld      s1, -128(s0)
    vl8r.v  v8, (s1) # Reload LMUL=8
    vadd.vv v8, v8, v24
    ld      s1, -136(s0)
    vl8r.v  v24, (s1) # Reload LMUL=8
    ld      s1, -144(s0)
    vl8r.v  v0, (s1) # Reload LMUL=8
    vadd.vv v24, v0, v24
    ld      s1, -128(s0)
    vs8r.v  v24, (s1) # Spill LMUL=8
    ld      s1, -152(s0)
    vl8r.v  v0, (s1) # Reload LMUL=8
    ld      s1, -160(s0)
    vl8r.v  v24, (s1) # Reload LMUL=8
    vadd.vv v0, v0, v24
    add     s1, a4, s10
    vse64.v v16, (s1)
    add     s1, s2, s10
    vse64.v v8, (s1)
    vadd.vv v8, v8, v16
    add     s1, t4, s10
    ld      s9, -128(s0)
    vl8r.v  v16, (s9) # Reload LMUL=8
    vse64.v v16, (s1)
    add     s9, t0, s10
    vadd.vv v8, v8, v16
    vle64.v v16, (s9)
    add     s1, t1, s10
    vse64.v v0, (s1)
    vadd.vv v8, v8, v0
    vsll.vi v16, v16, 1
    vadd.vv v8, v8, v16
    vse64.v v8, (s9)
    add     s6, s6, s7
    add     s10, s10, s8
    bne     s6, s4, .LBB0_4
```

If instead of using LMUL=1 for the 8-bit computation, the compiler is allowed to use a fractional LMUL=1/2, then the 64-bit computations can be performed using LMUL=4 (note that the same ratio of 64-bit elements and 8-bit elements is preserved as in the previous example). Now the compiler has 8 available

registers to perform register allocation, resulting in no spill code, as shown in the loop below:

```
.LBB0_4:                                # %vector.body
                                        # =>This Inner Loop Header: Depth=1
    add     s9, a2, s6
    vsetvli s1, zero, e8,mf2,ta,mu // LMUL=1/2 !
    vle8.v  v25, (s9)
    add     s1, a3, s6
    vle8.v  v26, (s1)
    vadd.vv v25, v26, v25
    add     s1, a1, s6
    vse8.v  v25, (s1)
    add     s9, a5, s10
    vsetvli s1, zero, e64,m4,ta,mu // LMUL=4
    vle64.v v28, (s9)
    add     s1, a6, s10
    vle64.v v8, (s1)
    vadd.vv v28, v8, v28
    add     s1, a7, s10
    vle64.v v8, (s1)
    add s1, s3, s10
    vle64.v v12, (s1)
    add     s1, t6, s10
    vle64.v v16, (s1)
    add     s1, t5, s10
    vle64.v v20, (s1)
    add     s1, a4, s10
    vse64.v v28, (s1)
    vadd.vv v8, v12, v8
    vadd.vv v12, v20, v16
    add     s1, t2, s10
    vle64.v v16, (s1)
    add     s1, t3, s10
    vle64.v v20, (s1)
    add     s1, s2, s10
    vse64.v v8, (s1)
    add     s9, t4, s10
    vadd.vv v16, v20, v16
    add     s11, t0, s10
    vle64.v v20, (s11)
    vse64.v v12, (s9)
    add     s1, t1, s10
    vse64.v v16, (s1)
    vsll.vi v20, v20, 1
    vadd.vv v28, v8, v28
    vadd.vv v28, v28, v12
    vadd.vv v28, v28, v16
    vadd.vv v28, v28, v20
    vse64.v v28, (s11)
```

```
    add     s6, s6, s7
    add     s10, s10, s8
    bne     s6, s4, .LBB0_4
```

Appendix D: Calling Convention for Vector State (Not authoritative - Placeholder Only)

This Appendix is only a placeholder to help explain the conventions used in the code examples, and is not considered frozen or part of the ratification process. The official RISC-V psABI document is being expanded to specify the vector calling conventions.

In the RISC-V psABI, the vector registers `v0`-`v31` are all caller-saved. The `vl` and `vtype` CSRs are also caller-saved.

Procedures may assume that `vstart` is zero upon entry. Procedures may assume that `vstart` is zero upon return from a procedure call.

Application software should normally not write `vstart` explicitly. Any procedure that does explicitly write `vstart` to a nonzero value must zero `vstart` before either returning or calling another procedure.

The `vxrm` and `vxsat` fields of `vcsr` have thread storage duration.

Executing a system call causes all caller-saved vector registers (`v0`-`v31`, `vl`, `vtype`) and `vstart` to become unspecified.

This scheme allows system calls that cause context switches to avoid saving and later restoring the vector registers.

Most OSes will choose to either leave these registers intact or reset them to their initial state to avoid leaking information across process boundaries.

Appendix E: Bit Manipulation Extensions Assembly Code Examples

The following examples provide software optimization guidance.

E.1. strlen

The **orc.b** instruction allows for the efficient detection of **NUL** bytes in an XLEN-sized chunk of data:

- the result of **orc.b** on a chunk that does not contain any **NUL** bytes will be all-ones, and
- after a bitwise-negation of the result of **orc.b**, the number of data bytes before the first **NUL** byte (if any) can be detected by **ctz**/**clz** (depending on the endianness of data).

A full example of a **strlen** function, which uses these techniques and also demonstrates the use of it for unaligned/partial data, is the following:

```
#include <sys/asm.h>

    .text
    .globl strlen
    .type  strlen, @function
strlen:
    andi    a3, a0, (SZREG-1)   // offset
    andi    a1, a0, -SZREG      // align pointer
.Lprologue:
    li      a4, SZREG
    sub     a4, a4, a3          // XLEN - offset
    slli    a3, a3, 3           // offset * 8
    REG_L   a2, 0(a1)           // chunk
    /*
     * Shift the partial/unaligned chunk we loaded to remove the bytes
     * from before the start of the string, adding NUL bytes at the end.
     */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    srl     a2, a2 ,a3          // chunk >> (offset * 8)
#else
    sll     a2, a2, a3
#endif
    orc.b   a2, a2
    not     a2, a2
    /*
     * Non-NUL bytes in the string have been expanded to 0x00, while
     * NUL bytes have become 0xff.  Search for the first set bit
     * (corresponding to a NUL byte in the original chunk).
     */
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    ctz     a2, a2
#else
    clz     a2, a2
```

```
#endif
    /*
     * The first chunk is special: compare against the number of valid
     * bytes in this chunk.
     */
    srli    a0, a2, 3
    bgtu    a4, a0, .Ldone
    addi    a3, a1, SZREG
    li      a4, -1
    .align 2
    /*
     * Our critical loop is 4 instructions and processes data in 4 byte
     * or 8 byte chunks.
     */
.Lloop:
    REG_L   a2, SZREG(a1)
    addi    a1, a1, SZREG
    orc.b   a2, a2
    beq     a2, a4, .Lloop

.Lepilogue:
    not     a2, a2
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    ctz     a2, a2
#else
    clz     a2, a2
#endif
    sub     a1, a1, a3
    add     a0, a0, a1
    srli    a2, a2, 3
    add     a0, a0, a2
.Ldone:
    ret
```

E.2. strcmp

```
#include <sys/asm.h>

  .text
  .globl strcmp
  .type  strcmp, @function
strcmp:
  or    a4, a0, a1
  li    t2, -1
  and   a4, a4, SZREG-1
  bnez  a4, .Lsimpleloop

  # Main loop for aligned strings
```

```
.Lloop:
  REG_L a2, 0(a0)
  REG_L a3, 0(a1)
  orc.b t0, a2
  bne   t0, t2, .Lfoundnull
  addi  a0, a0, SZREG
  addi  a1, a1, SZREG
  beq   a2, a3, .Lloop

  # Words don't match, and no null byte in first word.
  # Get bytes in big-endian order and compare.
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
  rev8  a2, a2
  rev8  a3, a3
#endif
  # Synthesize (a2 >= a3) ? 1 : -1 in a branchless sequence.
  sltu a0, a2, a3
  neg  a0, a0
  ori  a0, a0, 1
  ret

.Lfoundnull:
  # Found a null byte.
  # If words don't match, fall back to simple loop.
  bne   a2, a3, .Lsimpleloop

  # Otherwise, strings are equal.
  li    a0, 0
  ret

  # Simple loop for misaligned strings
.Lsimpleloop:
  lbu   a2, 0(a0)
  lbu   a3, 0(a1)
  addi  a0, a0, 1
  addi  a1, a1, 1
  bne   a2, a3, 1f
  bnez  a2, .Lsimpleloop

1:
  sub   a0, a2, a3
  ret

.size   strcmp, .-strcmp
```

Appendix F: Historical Rationale for Extensions

This appendix contains the rationale for RISC-V ISA extensions at the time they were ratified. Unlike the ISA specification, this appendix is ordered chronologically, so as to convey the motivation and architectural reasoning underpinning each extension at the time of ratification. For extensions ratified prior to the conception of this appendix (ca. 2025), the rationale will be added over time. In cases where the rationale was not recorded, the authors and editors will synthesize it from the historical record.

F.1. "Zihintpause" Extension for Pause Hint

The PAUSE instruction hints to a hart that it should temporarily reduce its rate of execution. It is normally used to save energy and execution resources while polling, e.g. while waiting for a spinlock to become free.

Much of the debate surrounding this extension centered on whether a facility similar to x86's MONITOR/MWAIT should instead be provided. We concluded that, even if such a facility were to be defined for RISC-V, it would not supplant PAUSE. PAUSE is more appropriate when polling for non-memory events, when polling for multiple events, or when software does not know precisely what events it is polling for. (Perhaps surprisingly, the latter case is ubiquitous, in part because it is the mechanism expected by the Linux kernel's `cpu_relax` API.)

F.2. "Zicond" Extension for Integer Conditional Operations

Replacing unpredictable branches with conditional-select or conditional-move instructions can mitigate a class of costly branch mispredictions. Unfortunately, conditional-select instructions require three source operands. These instructions are a logical addition to ISAs that include three-source integer instructions for other reasons, but are too costly otherwise.

Some ISAs have instead furnished conditional-move instructions, which consume less encoding space and avoid the extra register read in simple microarchitectures. Unfortunately, in register-renamed microarchitectures, these instructions incur costs simlar to conditional select, or require additional microarchitectural structures and micro-op-issue constraints.

The Zicond extension was defined to solve the same problem as conditional select and conditional move, but with very little incremental cost for complex microarchitectures. It provides conditional-zero instructions, which read two source operands and, based upon the zeroness of the second operand, produce either the first operand or zero. These instructions can be used as part of a three-instruction sequence to synthesize conditional select. Several common conditional-execution idioms require only two instructions, as would be the case with conditional select or move, including conditional addition, subtraction, and bitwise AND, OR, and XOR.

Two conditional-zero instructions are included: one that writes zero if the comparand is zero, and one that does so if the comparand is nonzero. Variants that perform magnitude comparisons with zero were considered but ultimately excluded for insufficient quantitative justification.

F.3. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions

While compare-and-swap for XLEN wide data may be accomplished using LR/SC, the CAS atomic instructions scale better to highly parallel systems than LR/SC. Many lock-free algorithms, such as a lock-free queue, require manipulation of pointer variables. A simple CAS operation may not be sufficient to guard against what is commonly referred to as the ABA problem in such algorithms that manipulate pointer variables. To avoid the ABA problem, the algorithms associate a reference counter with the pointer variable and perform updates using a quadword compare and swap (of both the pointer and the counter).

The double and quadword CAS instructions support implementation of algorithms for ABA problem avoidance.

The CAS instruction supports the C++11 atomic compare and exchange operation.

F.4. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0

The A-extension offers atomic memory operation (AMO) instructions for *words*, *doublewords*, and *quadwords* (only for `AMOCAS`). The absence of atomic operations for subword data types necessitates emulation strategies. For bitwise operations, this emulation can be performed via word-sized bitwise AMO* instructions. For non-bitwise operations, emulation is achievable using word-sized `LR`/`SC` instructions.

Several limitations arise from this emulation approach:

1. In systems with large-scale or Non-Uniform Memory Access (NUMA) configurations, emulation based on `LR`/`SC` introduces issues related to scalability and fairness, particularly under conditions of high contention.
2. Emulation of narrower AMOs through wider AMO* instructions on non-idempotent IO memory regions may result in unintended side effects.
3. Utilizing wider AMO* instructions for emulating narrower AMOs risks activating extraneous breakpoints or watchpoints.
4. In the absence of native support for subword atomics, compilers often resort to inlining code sequences to provide the required emulation. This practice contributes to an increase in code size, with consequent impacts on system performance and memory utilization.

The Zabha extension addresses these limitations by adding support for *byte* and *halfword* atomic memory operations to the RISC-V Unprivileged ISA.

Index

@

(calling convention
    standard), 154
(compressed
    formats), 154
(compressed. C.BREAKPOINTINSTR), 162
(compressed. C.CA), 161
(compressed. C.CR), 161
(compressed. C.DIINST), 162
(compressed. C.NOPINSTR), 162
(register source specifiers
    c-ext), 154

A

atomics
    misaligned, 85

B

bi-endian, 19

C

compressed
    C.ANDI, 161
    C.SRLI
        C.SRAI, 160
    cb-format load and store, 159
    CI, 160
    CIW, 160
    cj-format load and store, 158
    cr-format load and store, 158
    cs-format load and store, 157
    integer constant generation, 159
    integer register-immediate, 159
    register-based load and store, 157
core
    accelerator, 14
    cluster
        multiprocessors, 14
    component, 14
    extensions
        coprocessor, 14

D

decomposition, 86
design
    high performance, 85
    scalable, 85
double-precision
    floating point, 121
    to single-precision, 123

E

endian
    bi-, 19
    little and big, 19
exceptions, 20

F

FENCE, 85
FENCE.I
    finer-grained, 46
    forward compatibility, 46
    synchronization, 45
floating point
    convert and move, 123
    double precision, 121
    fused multiply-add, 116
    load and store, 122
floating-point
    classification, 119
    classify, 124
    compare, 124
    conversion, 117
    exception flag, 114
    requirements, 121
    supported precisions, 121

H

hart
    execution environment, 15
HINT
    PAUSE, 59

I

ILEN, 19
IMAFD, 19
interrupts, 20
ISA
    definition, 13

M

memory access
    implicit and explicit, 18, 18
MUL
    DIV, 65
    div by zero, 66
    DIVU, 65
    MULH, 65
    MULHSU, 65
    MULHU, 65
    Zmmul, 66

N

NaN
  generation, 115
  propagation, 115

O

operations
  memory, 85
  subnormal, 115

P

PAUSE
  duration, 59
  encoding, 59
  energy consumption, 59
  HINT, 59
  LR/RC sequences, 59

R

RV32E
  design, 40
  difference from RV32I, 40
RV64I
  compares, 41
  HINT, 43
  LD, 43
  LUI, 42
  RV64I-only, 41
  shifts, 41
  SLLI, 41
  SRAIW, 41
  SRLIW, 41
RV64I-only
  ADDW, 42
  SLLW, 42
  SRAW, 42
  SRLW, 42
  SUBW, 42
RVWMO, 85

S

SFENCE, 85
single-precision
  to double-precision, 123
store instruction word
  not included, 45

T

tininess
  handling, 115
traps, 20

U

unspecified
  behaviors, 21
  values, 21

Bibliography

RISC-V ELF psABI Specification. github.com/riscv/riscv-elf-psabi-doc/ .

RISC-V Assembly Programmer's Manual. github.com/riscv/riscv-asm-manual .

SAIL ISA Specification Language. github.com/rems-project/sail

ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic. (2008). "Institute of Electrical and Electronic Engineers".

GB/T 32905-2016: SM3 Cryptographic Hash Algorithm. (2016). Also GM/T 0004-2012. Standardization Administration of China. www.gmbz.org.cn/upload/2018-07-24/1532401392982079739.pdf

GB/T 32907-2016: SM4 Block Cipher Algorithm. (2016). Also GM/T 0002-2012. Standardization Administration of China. www.gmbz.org.cn/upload/2018-04-04/1522788048733065051.pdf

AMD. (2017). *AMD Random Number Generator*. Advanced Micro Devices. www.amd.com/system/files/TechDocs/amd-random-number-generator.pdf

Amdahl, G. M., Blaauw, G. A., & F. P. Brooks, J. (1964). Architecture of the IBM System/360. *IBM Journal of R. & D.*, *8*(2).

Anderson, R. J. (2020). *Security engineering - a guide to building dependable distributed systems (3. ed.)*. Wiley. www.cl.cam.ac.uk/ rja14/book.html

Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J., & Tokita, T. (2000). Camellia: A 128-bit block cipher suitable for multiple platforms—design andanalysis. *International Workshop on Selected Areas in Cryptography*, 39–56.

ARM. (2017). *ARM TrustZone True Random Number Generator: Technical Reference Manual*. ARM. infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100976_0000_00_en

Bak, P. (1986). The Devil's Staircase. *Phys. Today*, *39*(12), 38–45. doi.org/10.1063/1.881047

Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., & Regazzoni, F. (2015). Midori: A block cipher for low energy. *International Conference on the Theory and Application of Cryptology and Information Security*, 411–436.

Banik, S., Pandey, S. K., Peyrin, T., Sasaki, Y., Sim, S. M., & Todo, Y. (2017). GIFT: a small present. *International Conference on Cryptographic Hardware and Embedded Systems*, 321–345.

Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., & Tsay, J.-K. (2012). Efficient Padding Oracle Attacks on Cryptographic Hardware. In R. Safavi-Naini & R. Canetti (Eds.), *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (Vol. 7417, pp. 608–625). Springer. doi.org/10.1007/978-3-642-32009-5_36

Barker, E., & Kelsey, J. (2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. NIST Special Publication SP 800-90A Revision 1. doi.org/10.6028/NIST.SP.800-90Ar1

Barker, E., Kelsey, J., Roginsky, A., Turan, M. S., Buller, D., & Kaufer, A. (2021). *Recommendation for Random Bit Generator (RBG) Constructions*. Draft NIST Special Publication SP 800-90C.

Baudet, M., Lubicz, D., Micolod, J., & Tassiaux, A. (2011). On the Security of Oscillator-Based Random Number Generators. *J. Cryptology*, *24*(2), 398–425. doi.org/10.1007/s00145-010-9089-3

Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., & Sim, S. M. (2016). The SKINNY family of block ciphers and its low-latency variant MANTIS. *Annual International Cryptology Conference*, 123–153.

Blum, L., Blum, M., & Shub, M. (1986). A Simple Unpredictable Pseudo-Random Number Generator. *SIAM J. Comput.*, *15*(2), 364–383. doi.org/10.1137/0215025

Blum, M. (1986). Independent unbiased coin flips from a correlated biased source – A finite state Markov chain. *Combinatorica*, *6*(2), 97–108. doi.org/10.1007/BF02579167

Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y., & Vikkelsoe, C. (2007). PRESENT: An ultra-lightweight block cipher. *International Workshop on Cryptographic Hardware and Embedded Systems*, 450–466.

Buchholz, W. (1962). *Planning a computer system: Project Stretch*. McGraw-Hill Book Company.

Criteria, C. (2017). *Common Methodology for Information Technology Security Evaluation: Evaluation methodology*. Specification: Version 3.1 Revision 5. commoncriteriaportal.org/cc/

Dworkin, M. (2007). *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication SP 800-38D. doi.org/10.6028/NIST.SP.800-38D

Evtyushkin, D., & Ponomarev, D. V. (2016). Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, & S. Halevi (Eds.), *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (pp. 843–857). ACM. doi.org/10.1145/2976749.2978374

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., & Hennessy, J. (1990). Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, 15–26.

Grover, L. K. (1996). A Fast Quantum Mechanical Algorithm for Database Search. *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 212–219. doi.org/10.1145/237814.237866

Hajimiri, A., & Lee, T. H. (1998). A general theory of phase noise in electrical oscillators. *IEEE Journal of Solid-State Circuits*, *33*(2), 179–194. doi.org/10.1109/4.658619

Hajimiri, A., Limotyrakis, S., & Lee, T. H. (1999). Jitter and phase noise in ring oscillators. _ IEEE Journal of Solid-State Circuits_, *34*(6), 790–804. doi.org/10.1109/4.766813

Hamburg, M., Kocher, P., & Marson, M. E. (2012). *Analysis of Intel's Ivy Bridge Digital Random Number Generator*. Technical Report, Cryptography Research (Prepared for Intel).

Heil, T. H., & Smith, J. E. (1996). *Selective Dual Path Execution*. University of Wisconsin - Madison.

Hurley-Smith, D., & Hernández-Castro, J. C. (2020). Quantum Leap and Crash: Searching and Finding Bias in Quantum Random Number Generators. *ACM Transactions on Privacy and Security*, *23*(3), 1–25. doi.org/10.1145/3403643

ISO. (2016). *Information technology – Security techniques – Testing methods for the mitigation of non-invasive attack classes against cryptographic modules* (Standard ISO/IEC 17825:2016; Issue ISO/IEC 17825:2016). International Organization for Standardization.

ISO/IEC. (2018). *IT Security techniques – Hash-functions – Part 3: Dedicated hash-functions*. ISO/IEC Standard 10118-3:2018.

ISO/IEC. (2018). *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers. Amendment 2: SM4*. ISO/IEC Standard 18033-3:2010/DAmd 2 (en).

ITU. (2019). *Quantum noise random number generator architecture*. International Telecommunications Union. www.itu.int/rec/T-REC-X.1702-201911-I/en

Jaques, S., Naehrig, M., Roetteler, M., & Virdia, F. (2020). Implementing Grover Oracles for Quantum Key Search on AES and LowMC. In A. Canteaut & Y. Ishai (Eds.), *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II* (Vol. 12106, pp. 280–310). Springer. doi.org/10.1007/978-3-030-45724-2_10

Karaklajic, D., Schmidt, J.-M., & Verbauwhede, I. (2013). Hardware Designer's Guide to Fault Attacks. *IEEE Trans. Very Large Scale Integr. Syst.*, *21*(12), 2295–2306. doi.org/10.1109/TVLSI.2012.2231707

Katevenis, M. G. H., Sherburne, R. W., Jr., Patterson, D. A., & Séquin, C. H. (1983, August). The RISC II micro-architecture. *Proceedings VLSI 83 Conference*.

Killmann, W., & Schindler, W. (2001). *A Proposal for: Functionality classes and evaluation methodology for true (physical) random number generators*. BSI. www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_Functionality_classes_evaluation_methodology_for_true_RNG_e.html

Killmann, W., & Schindler, W. (2011). *A Proposal for: Functionality classes for random number generators*. BSI. www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_31_Functionality_classes_for_random_number_generators_e.html

Kim, H., Mutlu, O., Stark, J., & Patt, Y. N. (2005). Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution. *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 43–54.

Klauser, A., Austin, T., Grunwald, D., & Calder, B. (1998). Dynamic Hammock Predication for Non-Predicated Instruction Set Architectures. *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*.

Kwon, D., Kim, J., Park, S., Sung, S. H., Sohn, Y., Song, J. H., Yeom, Y., Yoon, E.-J., Lee, S., Lee, J., & others. (2003). New block cipher: ARIA. *International Conference on Information Security and Cryptology*, 432–445.

Lacharme, P. (2008). Post-Processing Functions for a Biased Physical Random Number Generator. In K. Nyberg (Ed.), *Fast Software Encryption, 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers* (Vol. 5086, pp. 334–342). Springer. doi.org/10.1007/978-3-540-71039-4_21

Lee, D. D., Kong, S. I., Hill, M. D., Taylor, G. S., Hodges, D. A., Katz, R. H., & Patterson, D. A. (1989). A VLSI Chip Set for a Multiprocessor Workstation—Part I: An RISC Microprocessor with Coprocessor Interface and Support for Symbolic Processing. *IEEE JSSC*, *24*(6), 1688–1698.

Lee, R. B., Shi, Z. J., Yin, Y. L., Rivest, R. L., & Robshaw, M. J. B. (2004). On permutation operations in cipher design. *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004.*, *2*, 569–577.

Liberty, J. S., Barrera, A., Boerstler, D. W., Chadwick, T. B., Cottier, S. R., Hofstee, H. P., Rosser, J. A., & Tsai, M. L. (2013). True hardware random number generation implemented in the 32-nm SOI POWER7+ processor. *IBM J. Res. Dev.*, *57*(6). doi.org/10.1147/JRD.2013.2279599

Markettos, A. T., & Moore, S. W. (2009). The Frequency Injection Attack on Ring-Oscillator-Based True

Random Number Generators. In C. Clavier & K. Gaj (Eds.), *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings* (Vol. 5747, pp. 317–331). Springer. doi.org/10.1007/978-3-642-04138-9_23

Marshall, B., Newell, G. R., Page, D., Saarinen, M.-J. O., & Wolf, C. (2020). The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, *2021*(1), 109–136. doi.org/10.46586/tches.v2021.i1.109-136

Marshall, B., Page, D., & Pham, T. (2019). *XCrypto: a cryptographic ISE for RISC-V* (No.1.0.0; Issue 1.0.0). github.com/scarv/xcrypto

Mechalas, J. P. (2018). *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*. Intel Technical Report, Version 2.1. software.intel.com/content/www/us/en/develop/articles/intel-digital-random-number-generator-drng-software-implementation-guide.html

Michael, M. M., & Scott, M. L. (1996). Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, 267–275. doi.org/10.1145/248052.248106

Moghimi, D., Sunar, B., Eisenbarth, T., & Heninger, N. (2020). TPM-FAIL: TPM meets Timing and Lattice Attacks. *29th USENIX Security Symposium (USENIX Security 20)*, To appear. www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm

Müller, S. (2020). *Documentation and Analysis of the Linux Random Number Generator, Version 3.6*. Prepared for BSI by atsec information security GmbH. www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN.pdf

NIST. (2001). *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication FIPS 197. doi.org/10.6028/NIST.FIPS.197

NIST. (2013). *Digital Signature Standard (DSS)*. Federal Information Processing Standards Publication FIPS 186-4. doi.org/10.6028/NIST.FIPS.186-4

NIST. (2015). *Secure Hash Standard (SHS)*. Federal Information Processing Standards Publication FIPS 180-4. doi.org/10.6028/NIST.FIPS.180-4

NIST. (2015). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. Federal Information Processing Standards Publication FIPS 202. doi.org/10.6028/NIST.FIPS.202

NIST. (2016). *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. Official Call for Proposals, National Institute for Standards and Technology. csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-final-dec-2016.pdf

NIST. (2019). *Security Requirements for Cryptographic Modules*. Federal Information Processing Standards Publication FIPS 140-3. doi.org/10.6028/NIST.FIPS.140-3

NIST, & CCCS. (2021). *Implementation Guidance for FIPS 140-3 and the Cryptographic Module Validation Program*. CMVP. csrc.nist.gov/CSRC/media/Projects/cryptographic-module-validation-program/documents/fips%20140-3/FIPS%20140-3%20IG.pdf

NSA/CSS. (2015). *Commercial National Security Algorithm Suite*. apps.nsa.gov/iaarchive/programs/iad-initiatives/cnsa-suite.cfm

Pan, H., Hindman, B., & Asanović, K. (2009, March). Lithe: Enabling Efficient Composition of Parallel Libraries. *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*.

Pan, H., Hindman, B., & Asanović, K. (2010, June). Composing Parallel Software Efficiently with Lithe. *31st Conference on Programming Language Design and Implementation*.

Patterson, D. A., & Séquin, C. H. (1981). RISC I: A Reduced Instruction Set VLSI Computer. *ISCA*, 443–458.

Rajwar, R., & Goodman, J. R. (2001). Speculative lock elision: enabling highly concurrent multithreaded execution. *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 294–305.

Rambus. (2020). *TRNG-IP-76 / EIP-76 Family of FIPS Approved True Random Generators*. Commercial Crypto IP. Formerly (2017) available from Inside Secure. www.rambus.com/security/crypto-accelerator-hardware-cores/basic-crypto-blocks/trng-ip-76/

Roux, P. (2014). Innocuous Double Rounding of Basic Arithmetic Operations. *Journal of Formalized Reasoning*, *7*(1), 131–142. doi.org/10.6092/issn.1972-5787/4359

Saarinen, M.-J. O. (2020). *Lightweight SHA ISA*. github.com/mjosaarinen/lwsha_isa .

Saarinen, M.-J. O. (2020). *Lightweight AES ISA*. github.com/mjosaarinen/lwaes_isa .

Saarinen, M.-J. O. (2021). *On Entropy and Bit Patterns of Ring Oscillator Jitter*. Preprint. arxiv.org/abs/2102.02196

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, 124–134. doi.org/10.1109/SFCS.1994.365700

Sinharoy, B., Kalla, R., Starke, W. J., Le, H. Q., Cargnoni, R., Van Norstrand, J. A., Ronchetti, B. J., Stuecheli, J., Leenstra, J., Guthrie, G. L., Nguyen, D. Q., Blaner, B., Marino, C. F., Retter, E., & Williams, P. (2011). IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, *55*(3), 1–1.

Suzaki, T., Minematsu, K., Morioka, S., & Kobayashi, E. (2012). TWINE: A Lightweight Block Cipher for Multiple Platforms. *International Conference on Selected Areas in Cryptography*, 339–354.

Thornton, J. E. (1965). Parallel Operation in the Control Data 6600. *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, 33–40.

Tremblay, M., Chan, J., Chaudhry, S., Conigliaro, A. W., & Tse, S. S. (2000). The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, *20*(6), 12–25.

Tseng, J., & Asanović, K. (2000). Energy-Efficient Register Access. *Proc. of the 13th Symposium on Integrated Circuits and Systems Design*, 377–384.

Turan, M. S., Barker, E., Kelsey, J., McKay, K. A., Baish, M. L., & Boyle, M. (2018). *Recommendation for the Entropy Sources Used for Random Bit Generation*. NIST Special Publication SP 800-90B. doi.org/10.6028/NIST.SP.800-90B

Ungar, D., Blau, R., Foley, P., Samples, D., & Patterson, D. (1984). Architecture of SOAR: Smalltalk on a RISC. *ISCA*, 188–197.

Valtchanov, B., Fischer, V., Aubert, A., & Bernard, F. (2010). Characterization of randomness sources in ring oscillator-based true random number generators in FPGAs. In E. Gramatová, Z. Kotásek, A. Steininger, H. T. Vierhaus, & H. Zimmermann (Eds.), *13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2010, Vienna, Austria, April 14-16, 2010* (pp. 48–53). IEEE Computer Society. doi.org/10.1109/DDECS.2010.5491819

Varchola, M., & Drutarovský, M. (2010). New High Entropy Element for FPGA Based True Random Number Generators. In S. Mangard & F.-X. Standaert (Eds.), *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings* (Vol. 6225, pp. 351–365). Springer. doi.org/10.1007/978-3-642-15031-9_24

von Neumann, J. (1951). Various Techniques Used in Connection with Random Digits. In A. S. Householder, G. E. Forsythe, & H. H. Germond (Eds.), *Monte Carlo Method* (Vol. 12, pp. 36–38). US Government Printing Office. mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf

Waterman, A. (2011). *Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed* (Issue UCB/EECS-2011-63) [Master's thesis]. University of California, Berkeley.

Waterman, A. (2016). *Design of the RISC-V Instruction Set Architecture* (Issue UCB/EECS-2016-1) [PhD thesis]. University of California, Berkeley.

Waterman, A., Lee, Y., Patterson, D. A., & Asanović, K. (2011). *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA* (UCB/EECS-2011-62; Issue UCB/EECS-2011-62). EECS Department, University of California, Berkeley.

Waterman, A., Lee, Y., Patterson, D. A., & Asanović, K. (2014). *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Version 2.0* (UCB/EECS-2014-54; Issue UCB/EECS-2014-54). EECS Department, University of California, Berkeley.

Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., & Verbauwhede, I. (2015). RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, *58*(12), 1–15.

RISC-V®