



DRAFT

The RISC-V Instruction Set Manual

Version 20260330: DRAFT---NOT AN OFFICIAL RELEASE

Table of Contents

Preamble	1
I: The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture	2
Preface	3
1. Introduction	14
1.1. RISC-V Hardware Platform Terminology	15
1.2. RISC-V Software Execution Environments and Harts	15
1.3. RISC-V ISA Overview	16
1.4. Memory	18
1.5. Base Instruction-Length Encoding	19
1.6. Exceptions, Traps, and Interrupts	21
1.7. UNSPECIFIED Behaviors and Values	22
2. RV32I Base Integer Instruction Set, Version 2.1	24
2.1. Programmers' Model for Base Integer ISA	24
2.2. Base Instruction Formats	26
2.3. Immediate Encoding Variants	27
2.4. Integer Computational Instructions	28
2.5. Control Transfer Instructions	31
2.6. Load and Store Instructions	34
2.7. Memory Ordering Instructions	36
2.8. Environment Call and Breakpoints	37
2.9. HINT Instructions	38
3. RV32E and RV64E Base Integer Instruction Sets, Version 2.0	41
3.1. RV32E and RV64E Programmers' Model	41
3.2. RV32E and RV64E Instruction Set Encoding	41
4. RV64I Base Integer Instruction Set, Version 2.1	42
4.1. Register State	42
4.2. Integer Computational Instructions	42
4.3. Load and Store Instructions	43
4.4. HINT Instructions	44
5. "Zi" Scalar Integer Extensions	46
5.1. "Zifencei" Extension for Instruction-Fetch Fence, Version 2.0	46
5.2. Ziccif Extension for Instruction-Fetch Atomicity, Version 1.0	47
5.3. Ziccrse Extension for Main Memory Reservability, Version 1.0	48
5.4. Ziccamoa Extension for Main Memory Atomics, Version 1.0	48
5.5. Ziccamoc Extension for Main Memory Compare-and-Swap, Version 1.0	49
5.6. Zicclsm Extension for Main Memory Misaligned Accesses, Version 1.0	49
5.7. Zic64b Extension for 64-byte Cache Blocks, Version 1.0	49
6. "Zicsr" Extension for Control and Status Register (CSR) Instructions, Version 2.0	50
6.1. CSR Instructions	50
7. "Zicntr" and "Zihpm" Extensions for Counters, Version 2.0	54
7.1. "Zicntr" Extension for Base Counters and Timers	54
7.2. "Zihpm" Extension for Hardware Performance Counters	56
8. "Zihintntl" Extension for Non-Temporal Locality Hints, Version 1.0	58

9. "Zhintpause" Extension for Pause Hint, Version 2.0	62
10. "Zimop" Extension for May-Be-Operations, Version 1.0	63
10.1. "Zcmop" Compressed May-Be-Operations Extension, Version 1.0	64
11. "Zicond" Extension for Integer Conditional Operations, Version 1.0.0	65
11.1. Instructions (in alphabetical order)	65
11.2. Usage examples	66
12. "M" Extension for Integer Multiplication and Division, Version 2.0	68
12.1. Multiplication Operations	68
12.2. Division Operations	68
12.3. Zmmul Extension, Version 1.0	69
13. "A" Extension for Atomic Instructions, Version 2.1	70
13.1. Specifying Ordering of Atomic Instructions	70
13.2. "Zalrsc" Extension for Load-Reserved/Store-Conditional Instructions	70
13.3. Eventual Success of Store-Conditional Instructions	73
13.4. "Zaamo" Extension for Atomic Memory Operations	75
14. "Za" Atomic Instructions	77
14.1. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions, Version 1.0.0	77
14.2. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0	81
14.3. "Zalasr" Atomic Load-Acquire and Store-Release Instructions, Version 1.0	82
14.4. Za128rs Extension for Reservation-Set Size, Version 1.0	85
14.5. Za64rs Extension for Reservation-Set Size, Version 1.0	85
14.6. "Zawrs" Extension for Wait-on-Reservation-Set instructions, Version 1.01	85
14.7. Zama16b Extension for 16-byte Misaligned Atomicity, Version 1.0	86
15. RVWMO Memory Consistency Model, Version 2.0	87
15.1. Definition of the RVWMO Memory Model	87
15.2. CSR Dependency Tracking Granularity	91
15.3. Source and Destination Register Listings	91
16. "Ztso" Extension for Total Store Ordering, Version 1.0	98
17. "CMO" Extensions for Base Cache Management Operation ISA, Version 1.0.0	99
17.1. Pseudocode for instruction semantics	99
17.2. Introduction	99
17.3. Background	99
17.4. Coherent Agents and Caches	101
17.5. CSR controls for CMO instructions	106
17.6. Extensions	108
17.7. Instructions	110
18. "F" Extension for Single-Precision Floating-Point, Version 2.2	114
18.1. F Register State	114
18.2. Floating-Point Control and Status Register	115
18.3. NaN Generation and Propagation	116
18.4. Subnormal Arithmetic	117
18.5. Single-Precision Load and Store Instructions	117
18.6. Single-Precision Floating-Point Computational Instructions	118
18.7. Single-Precision Floating-Point Conversion and Move Instructions	119
18.8. Single-Precision Floating-Point Compare Instructions	121

18.9. Single-Precision Floating-Point Classify Instruction.....	121
19. "D" Extension for Double-Precision Floating-Point, Version 2.2	123
19.1. D Register State	123
19.2. NaN Boxing of Narrower Values.....	123
19.3. Double-Precision Load and Store Instructions	124
19.4. Double-Precision Floating-Point Computational Instructions.....	124
19.5. Double-Precision Floating-Point Conversion and Move Instructions	125
19.6. Double-Precision Floating-Point Compare Instructions	126
19.7. Double-Precision Floating-Point Classify Instruction	126
20. "Q" Extension for Quad-Precision Floating-Point, Version 2.2	127
20.1. Quad-Precision Load and Store Instructions	127
20.2. Quad-Precision Computational Instructions.....	127
20.3. Quad-Precision Convert and Move Instructions	128
20.4. Quad-Precision Floating-Point Compare Instructions.....	128
20.5. Quad-Precision Floating-Point Classify Instruction	129
21. "Zfh" and "Zfhmin" Extensions for Half-Precision Floating-Point, Version 1.0	130
21.1. Half-Precision Load and Store Instructions	130
21.2. Half-Precision Computational Instructions	130
21.3. Half-Precision Conversion and Move Instructions.....	131
21.4. Half-Precision Floating-Point Compare Instructions	132
21.5. Half-Precision Floating-Point Classify Instruction.....	132
21.6. "Zfhmin" Standard Extension for Minimal Half-Precision Floating-Point.....	132
22. "BF16" Extensions for BFloat16-precision Floating-Point, Version 1.0	134
22.1. Introduction	134
22.2. Intended Audience.....	134
22.3. Number Format	135
22.4. Extensions	139
22.5. Instructions	141
Bibliography.....	146
23. "Zfa" Extension for Additional Floating-Point Instructions, Version 1.0	147
23.1. Load-Immediate Instructions.....	147
23.2. Minimum and Maximum Instructions	148
23.3. Round-to-Integer Instructions.....	149
23.4. Modular Convert-to-Integer Instruction	149
23.5. Move Instructions.....	149
23.6. Comparison Instructions	150
24. "Zfinx", "Zdinx", "Zhinx", "Zhinxmin" Extensions for Floating-Point in Integer Registers, Version 1.0	151
24.1. Processing of Narrower Values	151
24.2. Zdinx.....	151
24.3. Processing of Wider Values	152
24.4. Zhinx	152
24.5. Zhinxmin.....	152
24.6. Privileged Architecture Implications	152
25. "C" Extension for Compressed Instructions, Version 2.0	154

25.1. Overview.....	154
25.2. Compressed Instruction Formats.....	156
25.3. Load and Store Instructions.....	157
25.4. Control Transfer Instructions.....	160
25.5. Integer Computational Instructions.....	161
25.6. Usage of C Instructions in LR/SC Sequences.....	165
25.7. HINT Instructions.....	165
25.8. RVC Instruction Set Listings.....	166
26. "Zc*" Extension for Code Size Reduction, Version 1.0.0	169
26.1. Zc* Overview.....	169
26.2. C.....	170
26.3. Zce.....	170
26.4. MISA.C.....	170
26.5. Zca.....	171
26.6. Zcf (RV32 only).....	171
26.7. Zcd.....	171
26.8. Zcb.....	171
26.9. Zcmp.....	173
26.10. Zcmt.....	174
26.11. Zc instruction formats.....	174
26.12. Zcb instructions.....	175
26.13. PUSH/POP register instructions.....	187
26.14. Table Jump Overview.....	217
27. Bit Manipulation Extensions.....	223
27.1. "B" Extension for Bit Manipulation, Version 1.0.0.....	225
27.2. Zba: Extension for Address generation, Version 1.0.0.....	225
27.3. Zbb: Extension for Basic bit-manipulation, Version 1.0.0.....	225
27.4. Zbc: Extension for Carry-less multiplication, Version 1.0.0.....	227
27.5. Zbs: Extension for Single-bit instructions, Version 1.0.0.....	228
27.6. Zbkb: Extension for Bit-manipulation for Cryptography, Version 1.0.0.....	228
27.7. Zbkc: Extension for Carry-less multiplication for Cryptography, Version 1.0.0.....	229
27.8. Zbkx: Extension for Crossbar permutations, Version 1.0.0.....	229
27.9. Instructions (in alphabetical order).....	230
28. "V" Standard Extension for Vector Operations, Version 1.0.....	281
28.1. Introduction.....	281
28.2. Implementation-defined Constant Parameters.....	281
28.3. Vector Extension Programmer's Model.....	281
28.4. Mapping of Vector Elements to Vector Register State.....	290
28.5. Vector Instruction Formats.....	294
28.6. Configuration-Setting Instructions (vsetvli/vsetivli/vsetvl).....	299
28.7. Vector Loads and Stores.....	303
28.8. Vector Memory Alignment Constraints.....	317
28.9. Vector Memory Consistency Model.....	317
28.10. Vector Arithmetic Instruction Formats.....	318
28.11. Vector Integer Arithmetic Instructions.....	322

28.12. Vector Fixed-Point Arithmetic Instructions	334
28.13. Vector Floating-Point Instructions	337
28.14. Vector Reduction Operations	354
28.15. Vector Mask Instructions	357
28.16. Vector Permutation Instructions	364
28.17. Exception Handling	371
28.18. Standard Vector Extensions	373
28.19. Vector Element Groups	376
28.20. Vector Instruction Listing	378
29. Cryptography Extensions: Scalar & Entropy Source Instructions, Version 1.0.1	384
29.1. Introduction	384
29.2. Extensions Overview	386
29.3. Instructions	389
29.4. Entropy Source	439
29.5. Data Independent Execution Latency Subset: Zkt	442
29.6. Instruction Rationale	448
29.7. Entropy Source Rationale and Recommendations	452
29.8. Supplementary Materials	464
29.9. Supporting Sail Code	464
30. Cryptography Extensions: Vector Instructions, Version 1.0	472
30.1. Introduction	472
30.2. Extensions Overview	477
30.3. Instructions	496
30.4. Crypto Vector Cryptographic Instructions	550
30.5. Vector Bitmanip and Carry-less Multiply Instructions	551
30.6. Supporting Sail Code	554
31. "Zvknhk" Extension for Vector Keccak	563
31.1. Extension Overview	563
31.2. vkeccak.vi	564
31.3. Encodings	569
32. Control-flow Integrity (CFI)	572
32.1. Landing Pad (Zicfilp)	572
32.2. Shadow Stack (Zicfiss)	576
33. "Zilsd", "Zclsd" Extensions for Load/Store pair for RV32, Version 1.0	587
33.1. Load/Store pair instructions (Zilsd)	587
33.2. Compressed Load/Store pair instructions (Zclsd)	587
33.3. Use of x0 as operand	588
33.4. Exception Handling	588
33.5. Instructions	589
34. RV32/64G Instruction Set Listings	595
35. ISA Extension Naming Conventions	608
35.1. Case Sensitivity	608
35.2. Base Integer ISA	608
35.3. Instruction-Set Extension Names	608
35.4. Underscores	608

35.5. Additional Standard Unprivileged Extension Names	608
35.6. Supervisor-level Instruction-Set Extension Names	609
35.7. Hypervisor-level Instruction-Set Extension Names	609
35.8. Machine-level Instruction-Set Extension Names	609
35.9. Non-Standard Extension Names	609
35.10. Version Numbers	610
35.11. Subset Naming Convention	610
Appendix A: Historical Rationale for Extensions	612
A.1. "Zihintpause" Extension for Pause Hint	612
A.2. "Zicond" Extension for Integer Conditional Operations	612
A.3. "Zacas" Extension for Atomic Compare-and-Swap (CAS) Instructions	612
A.4. "Zabha" Extension for Byte and Halfword Atomic Memory Operations, Version 1.0	613
Appendix B: RVWMO Explanatory Material, Version 0.1	614
B.1. Why RVWMO?	614
B.2. Litmus Tests	614
B.3. Explaining the RVWMO Rules	616
B.4. Beyond Main Memory	629
B.5. Code Porting and Mapping Guidelines	631
B.6. Implementation Guidelines	636
B.7. Known Issues	639
Appendix C: Formal Memory Model Specifications, Version 0.1	641
C.1. Formal Axiomatic Specification in Alloy	641
C.2. Formal Axiomatic Specification in Herd	645
C.3. An Operational Memory Model	648
Appendix D: Vector Assembly Code Examples	663
D.1. Vector-vector add example	663
D.2. Example with mixed-width mask and compute	663
D.3. Memcpy example	664
D.4. Conditional example	664
D.5. SAXPY example	665
D.6. SGEMM example	665
D.7. Division approximation example	670
D.8. Square root approximation example	670
D.9. C standard library strcmp example	671
D.10. Fractional Lmul example	671
Appendix E: Bit Manipulation Extensions Assembly Code Examples	676
E.1. strlen	676
E.2. strcmp	677
II: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture	679
Preface	680
1. Introduction	688
1.1. RISC-V Privileged Software Stack Terminology	688
1.2. Privilege Levels	689
1.3. Debug Mode	690
2. Control and Status Registers (CSRs)	691

2.1. CSR Address Mapping Conventions.....	691
2.2. CSR Listing.....	694
2.3. CSR Field Specifications.....	703
2.4. CSR Field Modulation.....	704
2.5. Implicit Reads of CSRs.....	705
2.6. CSR Width Modulation.....	705
2.7. Explicit Accesses to CSRs Wider than XLEN.....	705
3. Machine-Level ISA, Version 1.13.....	706
3.1. Machine-Level CSRs.....	706
3.2. Machine-Level Memory-Mapped Registers.....	739
3.3. Machine-Mode Privileged Instructions.....	741
3.4. Reset.....	743
3.5. Non-Maskable Interrupts.....	744
3.6. Physical Memory Attributes.....	744
3.7. Physical Memory Protection.....	750
4. Supervisor-Level ISA, Version 1.13.....	755
4.1. Supervisor CSRs.....	755
4.2. Supervisor Instructions.....	770
4.3. Sv32: Page-Based 32-bit Virtual-Memory Systems.....	773
4.4. Sv39: Page-Based 39-bit Virtual-Memory System.....	779
4.5. Sv48: Page-Based 48-bit Virtual-Memory System.....	780
4.6. Sv57: Page-Based 57-bit Virtual-Memory System.....	781
5. "H" Extension for Hypervisor Support, Version 1.0.....	783
5.1. Privilege Modes.....	783
5.2. Hypervisor and Virtual Supervisor CSRs.....	784
5.3. Hypervisor Instructions.....	801
5.4. Machine-Level CSRs.....	803
5.5. Two-Stage Address Translation.....	807
5.6. Traps.....	810
6. "Sm" Machine Extensions.....	821
6.1. "Smstateen/Ssstateen" Extensions, Version 1.0.....	821
6.2. "Smcsrind/Sscsrind" Indirect CSR Access, Version 1.0.....	826
6.3. "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode, Version 1.0.....	830
6.4. "Smcntrpmf" Cycle and Instret Privilege Mode Filtering, Version 1.0.....	833
6.5. "Smrnmi" Extension for Resumable Non-Maskable Interrupts, Version 1.0.....	835
6.6. "Smcdeleg/Ssccfg" Counter Delegation Extensions, Version 1.0.....	837
6.7. "Smdbltrp" Double Trap Extension, Version 1.0.....	840
6.8. "Smctr" Control Transfer Records Extension, Version 1.0.....	840
6.9. Control-flow Integrity (CFI).....	855
6.10. Pointer Masking Extensions, Version 1.0.0.....	861
7. "Sv" Supervisor Virtual-Memory Extensions.....	869
7.1. "Svnapot" Extension for NAPOT Translation Contiguity, Version 1.0.....	869
7.2. "Svpbmt" Extension for Page-Based Memory Types, Version 1.0.....	870
7.3. "Svadu" Extension for Hardware Updating of A/D Bits, Version 1.0.....	872

7.4. "Svinval" Extension for Fine-Grained Address-Translation Cache Invalidation, Version 1.0	872
7.5. "Svvpctc" Extension for Obviating Memory-Management Instructions after Marking PTEs Valid, Version 1.0	874
7.6. "Svrsw60t59b" Extension for PTE Reserved-for-Software Bits 60-59, Version 1.0	874
8. "Ss" Supervisor Extensions	876
8.1. "Ssqosid" Extension for Quality-of-Service (QoS) Identifiers, Version 1.0	876
8.2. Ssu64xl Extension for UXLEN=64 Support, Version 1.0	877
8.3. Ssccptr Extension for Main Memory Page-Table Reads, Version 1.0	877
8.4. Sstvecd Extension for Direct Trap Vectoring, Version 1.0	878
8.5. Sstvala Extension for Trap Value Reporting, Version 1.0	878
8.6. Sscounterenw Extension for Counter-Enable Writability, Version 1.0	878
8.7. Ssstric Extension for Extension Conformance, Version 1.0	878
8.8. "Sstc" Extension for Supervisor-mode Timer Interrupts, Version 1.0	878
8.9. "Sscofpmf" Extension for Count Overflow and Mode-Based Filtering, Version 1.0	878
8.10. "Ssdbltrp" Double Trap Extension, Version 1.0	880
9. "Sh" Hypervisor Extensions	881
9.1. Shvstvecd Extension for Direct Trap Vectoring, Version 1.0	881
9.2. Shcounterenw Extension for Counter-Enable Writability, Version 1.0	881
9.3. Shvstvala Extension for Trap Value Reporting, Version 1.0	881
9.4. Shtvala Extension for Trap Value Reporting, Version 1.0	881
9.5. Shvsatpa Extension for Translation Mode Support, Version 1.0	881
9.6. Shgatpa Extension for Translation Mode Support, Version 1.0	881
9.7. Sha Augmented Hypervisor Extension	881
10. RISC-V Privileged Instruction Set Listings	883
Appendix A: Historical Rationale for Extensions	885
A.1. "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode	885
III: The RISC-V Instruction Set Manual, Volume III: Profiles	888
Preface	889
1. Introduction	890
1.1. Profiles versus Platforms	890
1.2. Components of a Profile	891
2. RVI20 Profiles	896
2.1. RVI20U32	896
2.2. RVI20U64	897
3. RVA20 Profiles	899
3.1. RVA20U64 Profile	899
3.2. RVA20S64 Profile	901
4. RVA22 Profiles	903
4.1. RVA22U64 Profile	903
4.2. RVA22S64 Profile	906
5. RVA23 Profiles	909
5.1. RVA23U64 Profile	909
5.2. RVA23S64 Profile	912
6. RVB23 Profiles	915

6.1. Introduction	915
6.2. RVB23 Profiles.....	915
Index.....	921
Bibliography.....	923

DRAFT

Preamble

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Arvind, Krste Asanović, Peter Ashenden, Derek Atkins, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Scott Beamer, Jonathan Behrens, Abel Bernabeu, Hans Boehm, Paolo Bonzini, Alex Bradbury, Preston Briggs, Ruslan Bukin, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Anthony Coulter, Palmer Dabbelt, Monte Dalrymple, L Peter Deutsch, Ken Dockser, Paul Donahue, Aaron Durbin, Roger Espasa, Greg Faver, Dennis Ferguson, Shaked Flur, Stefan Freudenberger, Mike Frysinger, Marc Gauthier, Andy Glew, Jan Gray, Gianluca Guida, Gary Guo, Michael Hamburg, John Hauser, Christian Herber, David Horner, Bruce Houlton, Bill Huffman, John Ingalls, Alexandre Joannou, Olof Johansson, Ben Keller, David Kruckemyer, Tariq Kurd, Yunsup Lee, Paul Loewenstein, Daniel Lustig, Andrew Lutomirski, Martin Maas, Yatin Manerkar, Luc Marangé, Ben Marshall, Margaret Martonosi, Phil McCoy, Nathan Menhorn, Christoph Müllner, Prashanth Mundkur, Joseph Myers, Vijayanand Nagarajan, Torbjørn Viem Ness, Jonathan Neuschäfer, Rishiyur Nikhil, Jonas Oberhauser, Stefan O'Rear, Albert Ou, John Ousterhout, Daniel Page, David Patterson, Dmitri Pavlov, Kade Phillips, Christopher Pulte, Jose Renau, Markku-Juhani O. Saarinen, 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, Steve Wallach, Paul Wamsley, Andrew Waterman, Robert Watson, David Weaver, Derek Williams, Claire Wolf, Andrew Wright, Adam Zabrocki, Reinoud Zandijk, and Sizhuo Zhang. This document is released under a Creative Commons Attribution 4.0 International License.

Please cite this document as: "The RISC-V Instruction Set Manual, Document Version 20260330-draft", Editors Andrew Waterman and Krste Asanović, RISC-V International, March 2026.

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. It is also a derivative of "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.9.1", released under the following license: ©2010-2017 Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson, Krste Asanović. Creative Commons Attribution 4.0 International License.

Part I: The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture

DRAFT

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
Zihpm	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	1.0	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

Base	Version	Status
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:

- 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
Zihintntnl	1.0	Ratified
Zihintpause	2.0	Ratified
Zimop	1.0	Ratified

Base	Version	Status
Zicond	1.0	Ratified
Zilsd	1.0	Ratified
M	2.0	Ratified
Zmmul	1.0	Ratified
A	2.1	Ratified
Zawrs	1.0	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

Base	Version	Status
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 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
Zihintntnl	1.0	Ratified
Zhintpause	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	1.0	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

Base	Version	Status
Zfh	1.0	Ratified
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 unrated 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 **fcscr**.
- 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, Section 3.3.3](#); 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 35](#).

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 $2^{\text{XLEN}}-1$ 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 [Section 5.1](#)).

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 15](#) and in appendices. Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in [Chapter 16](#). 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 25](#) 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 volume 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 the unprivileged architecture. 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.

DRAFT

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).

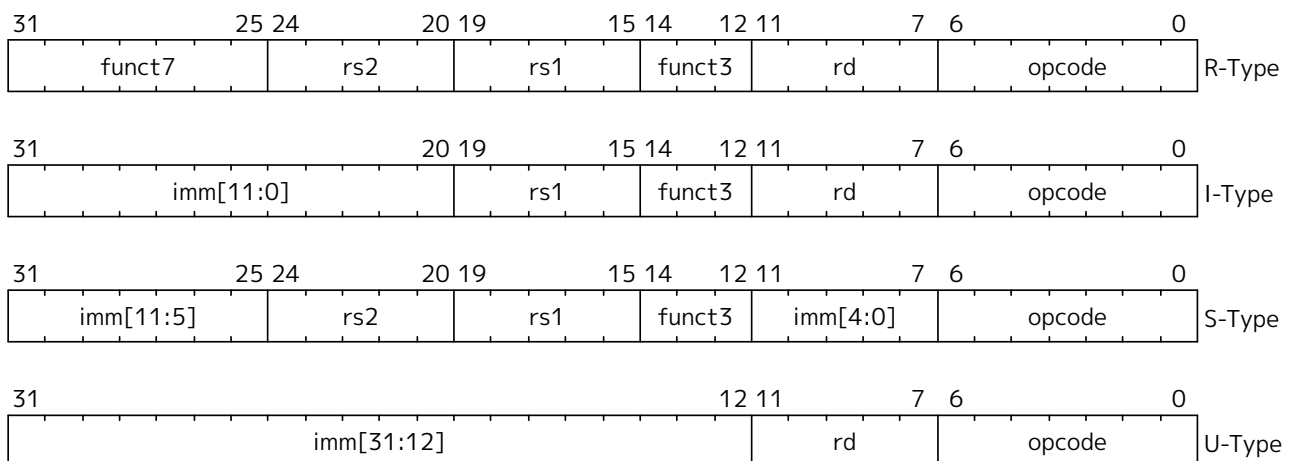
Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with IALIGN=32, where these are the only places where misalignment can occur.

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-

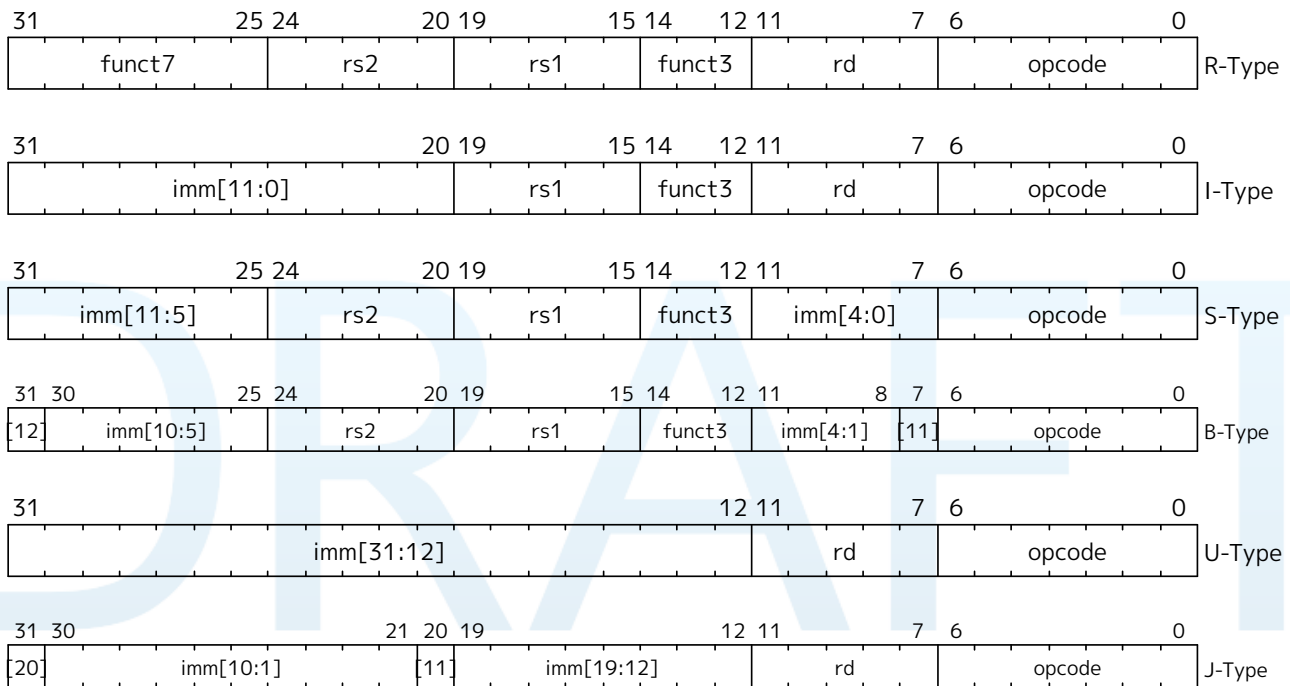
IV aka. SPUR ([Lee et al., 1989](#))).

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

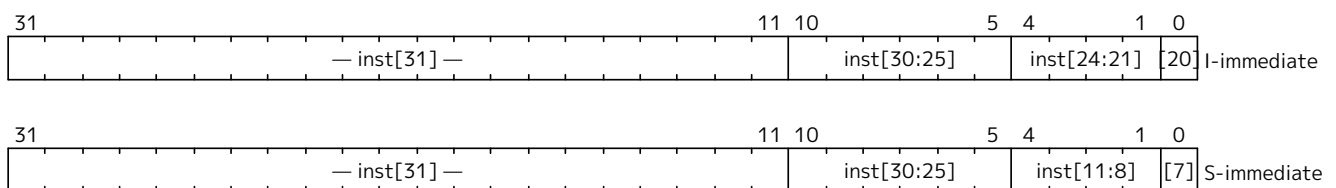
There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in [Base instruction formats immediate variants](#).



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.



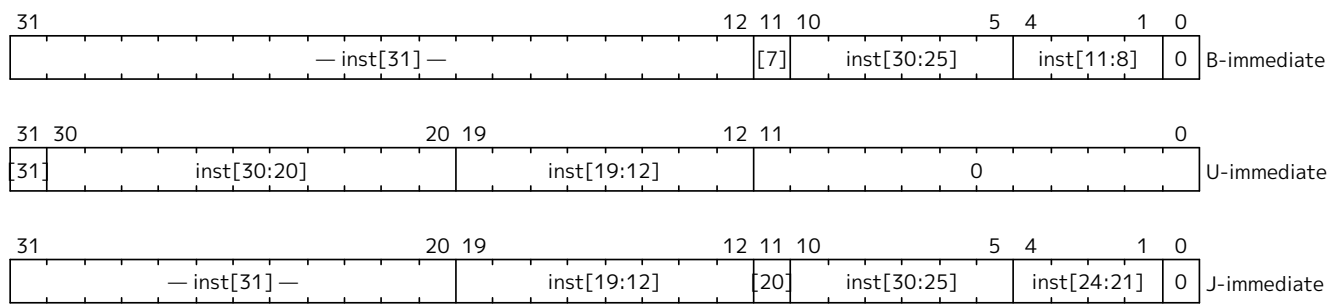


Figure 1. Types of immediate produced by RISC-V instructions.

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.



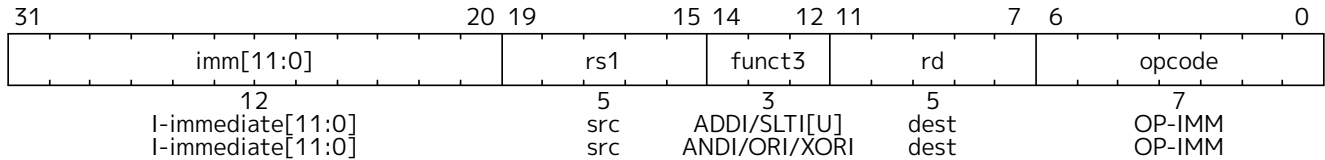
We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after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

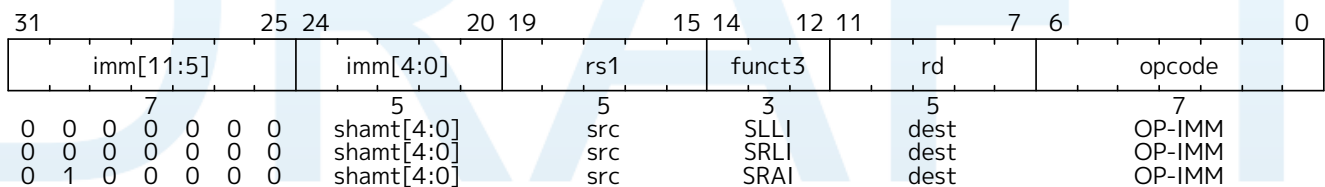

2.4.1. Integer Register-Immediate Instructions



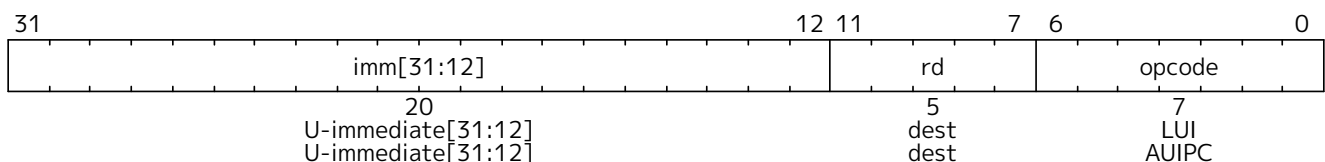
ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd*, *rs1*, *O* is used to implement the MV *rd*, *rs1* assembler pseudoinstruction.

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.

31								25 24		20 19		15 14		12 11		7 6		0	
funct7								rs2		rs1		funct3		rd		opcode			
7								5		5		3		5		7			
0	0	0	0	0	0	0	0	src2		src1		ADD/SLT[U]		dest		OP			
0	0	0	0	0	0	0	0	src2		src1		AND/OR/XOR		dest		OP			
0	0	0	0	0	0	0	0	src2		src1		SLL/SRL		dest		OP			
0	1	0	0	0	0	0	0	src2		src1		SUB/SRA		dest		OP			

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

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																						

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.

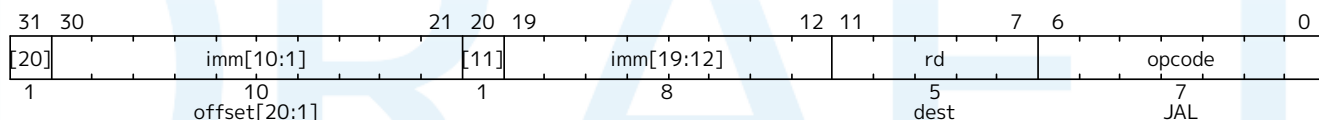
2.5.1. Unconditional Jumps

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.



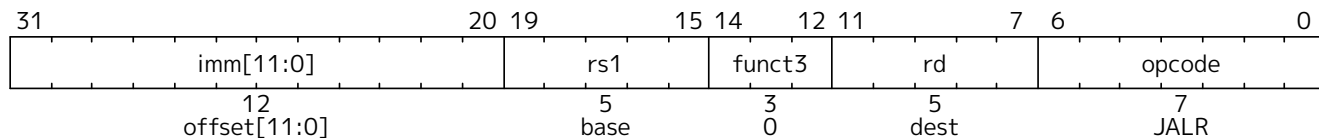
*The alternate link register supports calling millicode routines (e.g., those to save and restore registers in compressed code) while preserving the regular return address register. The register **x5** was chosen as the alternate link register as it maps to a temporary in the standard calling convention, 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

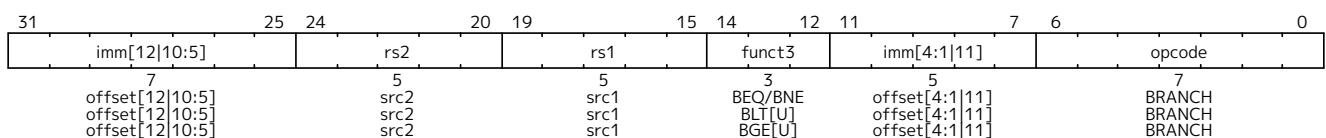
Some other ISAs added explicit hint bits to their indirect-jump instructions to guide return-address stack manipulation. We use implicit hinting tied to register numbers and the calling convention to reduce the encoding space used for these hints.



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.

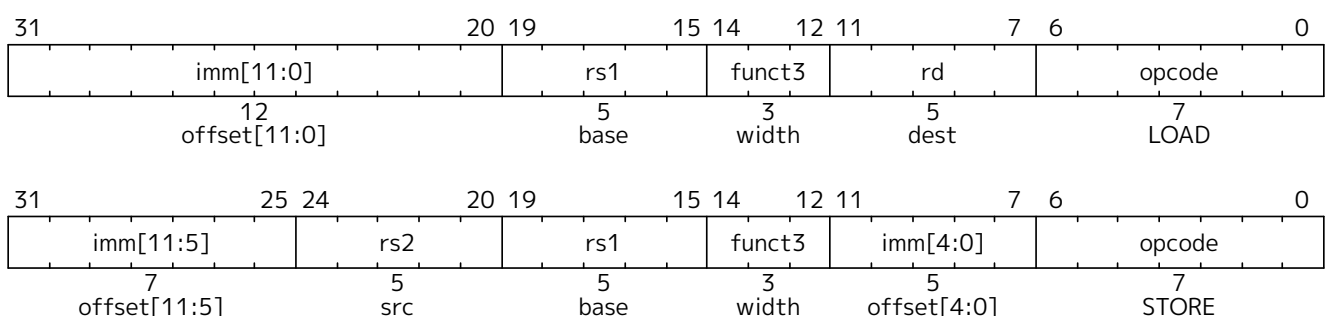
The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.



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.

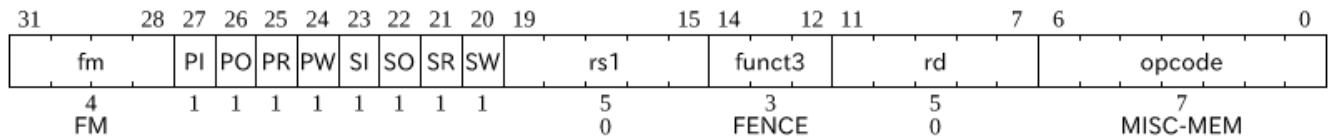
An EEI may not guarantee misaligned loads and stores are handled invisibly. In this case, loads and stores that are not naturally aligned may either complete execution successfully or raise an exception. The exception raised can be either an address-misaligned exception or an access-fault exception. 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 15](#) 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

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

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.



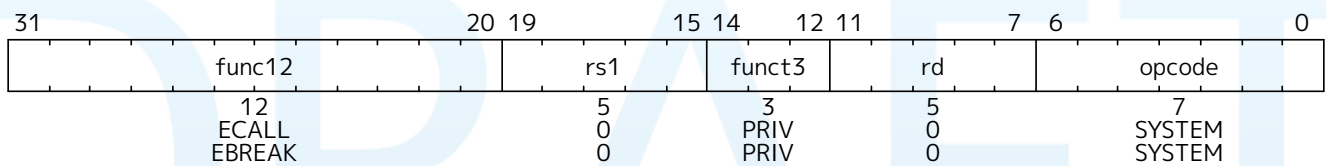
We chose a relaxed memory model to allow high performance from simple machine implementations and from likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative 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 **EEL** 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 "semlhosting", 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 25](#).

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}	<i>Designated for future standard use</i>
AUIPC	$rd=x0$	2^{20}	
ADDI	$rd=x0$, and either $rs1 \neq x0$ or $imm \neq 0$	$2^{17}-1$	
ANDI	$rd=x0$	2^{17}	
ORI	$rd=x0$	2^{17}	
XORI	$rd=x0$	2^{17}	
ADD	$rd=x0, rs1 \neq x0$	$2^{10}-32$	
ADD	$rd=x0, rs1=x0, rs2 \neq x2-x5$	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}	<i>Designated for future standard use</i>
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}	
FENCE	$rd=x0, rs1 \neq x0, fm=0$, and either $pred=0$ or $succ=0$	$2^{10}-63$	
FENCE	$rd \neq x0, rs1=x0, fm=0$, and either $pred=0$ or $succ=0$	$2^{10}-63$	
FENCE	$rd=rs1=x0, fm=0, pred=0, succ \neq 0$	15	
FENCE	$rd=rs1=x0, fm=0, pred \neq W, succ=0$	15	
FENCE	$rd=rs1=x0, fm=0, pred=W, succ=0$	1	PAUSE
SLTI	$rd=x0$	2^{17}	<i>Designated for custom use</i>
SLTIU	$rd=x0$	2^{17}	
SLLI	$rd=x0$, and either $rs1 \neq x0$ or $shamt \neq 31$	$2^{10}-1$	
SRLI	$rd=x0$	2^{10}	
SRAI	$rd=x0$, and either $rs1 \neq x0$ or $shamt \neq 7$	$2^{10}-1$	
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.

DRAFT

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.

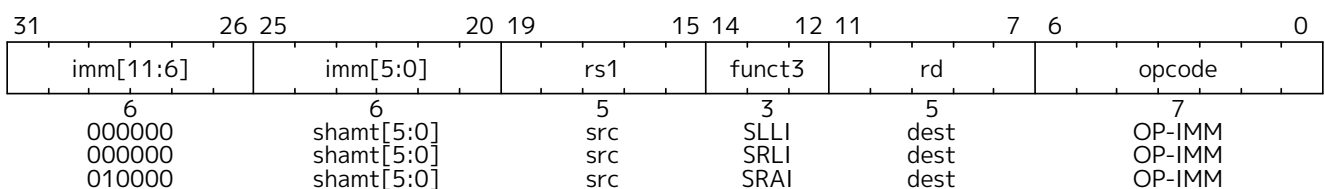


The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion 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.

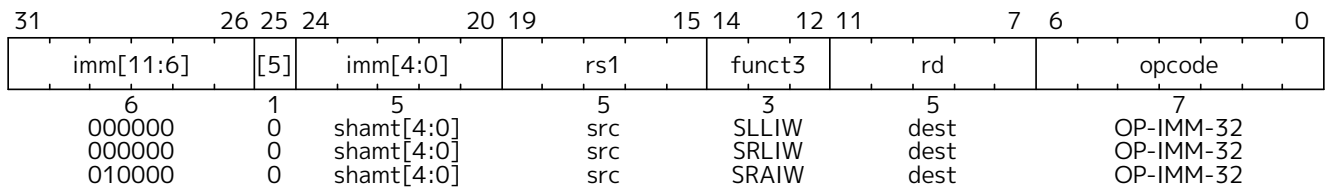
4.2.1. Integer Register-Immediate Instructions



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).



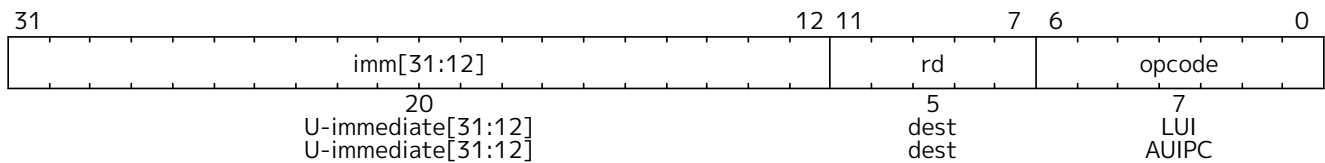
Shifts by a constant are encoded as a specialization of the I-type format using the same instruction opcode as RV32I. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the I-immediate field for RV64I. The right-shift type is encoded in bit 30. SLLI is a logical 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 $imm[5] \neq 0$ are reserved.



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



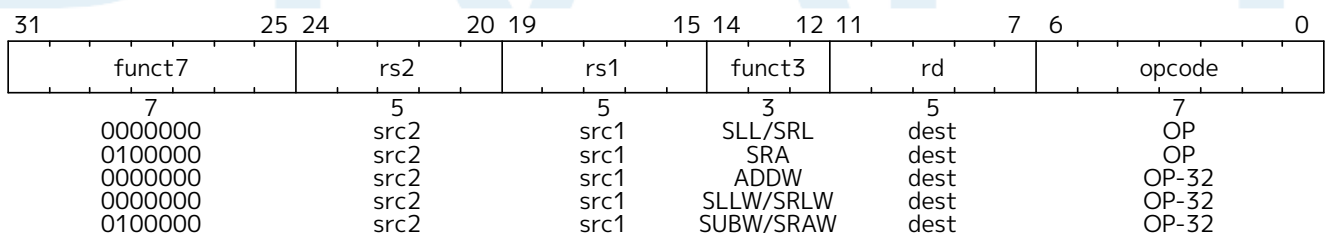
LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 32-bit U-immediate into register rd , filling in the lowest 12 bits with zeros. The 32-bit result is sign-extended to 64 bits.

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.

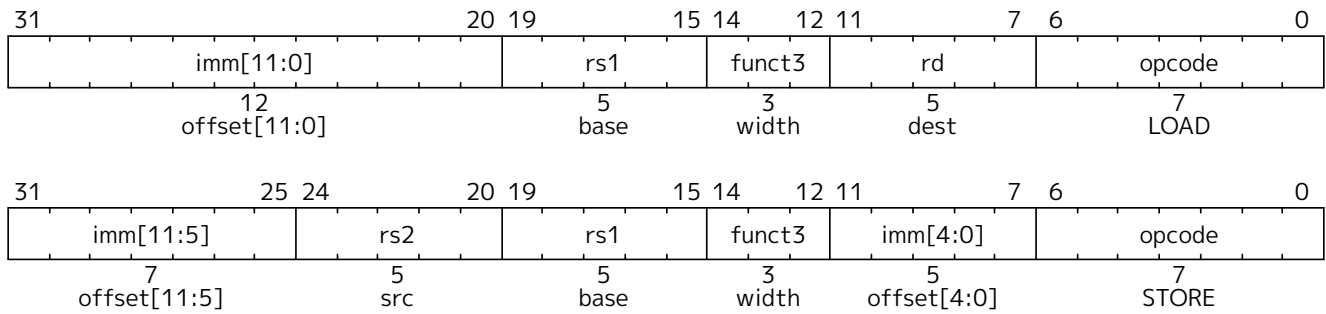
SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register $rs1$ by the shift amount held in register $rs2$. In RV64I, only the low 6 bits of $rs2$ are considered for the shift amount.

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 $rs1 \neq x0$ or $imm \neq 0$	$2^{17} - 1$	
ANDI	$rd = x0$	2^{17}	
ORI	$rd = x0$	2^{17}	
XORI	$rd = x0$	2^{17}	
ADDIW	$rd = x0$	2^{17}	
ADD	$rd = x0, rs1 \neq x0$	$2^{10} - 32$	
ADD	$rd = x0, rs1 = x0, rs2 \neq x2\_x5$	28	$(rs2 = x2)$ NTL.P1 $(rs2 = x3)$ NTL.PALL $(rs2 = x4)$ NTL.S1 $(rs2 = x5)$ NTL.ALL
ADD	$rd = x0, rs1 = x0, rs2 = x2\_x5$	4	
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	$rd = x0, rs1 \neq x0, fm = 0$, and either $pred = 0$ or $succ = 0$	$2^{10} - 63$	
FENCE	$rd \neq x0, rs1 = x0, fm = 0$, and either $pred = 0$ or $succ = 0$	$2^{10} - 63$	
FENCE	$rd = rs1 = x0, fm = 0, pred = 0, succ \neq 0$	15	PAUSE
FENCE	$rd = rs1 = x0, fm = 0, pred \neq W, succ = 0$	15	
FENCE	$rd = rs1 = x0, fm = 0, pred = W, succ = 0$	1	Designated for custom use
SLTI	$rd = x0$	2^{17}	
SLTIU	$rd = x0$	2^{17}	
SLLI	$rd = x0$, and either $rs1 \neq x0$ or $shamt \neq 31$	$2^{11} - 1$	
SRLI	$rd = x0$	2^{11}	
SRAI	$rd = x0$, and either $rs1 \neq x0$ or $shamt \neq 7$	$2^{11} - 1$	
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. "Zi" Scalar Integer Extensions



This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

5.1. "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.

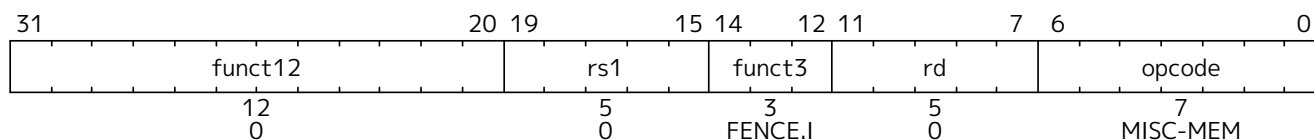
The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.



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:

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

Consumer hart:

```
la t2, flag
lw a0, (t2)
fence.i
patch_me:
c.li a1, 0
```

*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.

5.2. Ziccif Extension for Instruction-Fetch Atomicity, Version 1.0

This extension was ratified alongside the RVA20U64 profile. This chapter supplies an operational definition for the extension and adds expository material.

If the Ziccif extension is implemented, main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes of at most $\min(\text{ILEN}, \text{XLEN})$ bits are atomic.

An implementation with the Ziccif extension fetches instructions in a manner equivalent to the following state machine.

1. Let M be the smallest power of 2 such that $M \geq \min(\text{ILEN}, \text{XLEN})/8$. Let N be the pc modulo M . Atomically fetch $M - N$ bytes from memory at address pc . Let T be the running total of bytes fetched, initially $M - N$.
2. If the T bytes fetched begin with a complete instruction of length $L \leq T$, then execute that instruction, discard the remaining $T - L$ bytes fetched, and go back to step 1, using the updated pc . Otherwise, atomically fetch M bytes from memory at address $\text{pc} + T$, increment T by M , and repeat step 2.

The instruction-fetch atomicity rule supports concurrent code modification. When a hart modifies instruction memory and either it or another hart executes the modified instructions without first having executing a FENCE.I, the modifying hart should adhere to the following rules to ensure predictable behavior:

- *Modification stores must be single-copy atomic, hence must be naturally aligned.*
- *The modified instruction must not span an aligned M -byte boundary, unless it is replaced with a shorter unconditional control transfer (e.g., `c.ebreak` or `c.j`) that does not itself span an M -byte boundary.*
- *Modification stores must alter a complete instruction or complete instructions that do not collectively span an M -byte boundary, modulo the exception above that the first part of an instruction may be replaced with an unconditional control transfer instruction.*
- *Modifications must not combine smaller instructions into a larger instruction but may convert a larger instruction to some number of smaller instructions.*
- *Modified instruction memory must have the coherence PMA.*

Other well-defined code-modification strategies exist, but these rules provide a safe harbor.

Note that the software modifying the code need not know the value of M . Because ILEN must be at least the width of the instruction being modified, a lower bound on M can be inferred from the instruction's width and XLEN .

Memory protection and executability PMAs are applied only to bytes that are not discarded by this algorithm.

For example, if $M=8$, $N=0$, and the PMP granularity is 4 bytes, then it is valid to fetch a 4-byte instruction at pc , even if fetching from $\text{pc} + 4$ would have been disallowed by PMP.

For simplicity, implementations are likely to choose a PMP granularity no smaller than M .

5.3. Ziccrse Extension for Main Memory Reservability, Version 1.0

If the Ziccrse extension is implemented, then main memory regions with both the cacheability and coherence PMAs must support the RsrvEventual PMA.

5.4. Ziccamoa Extension for Main Memory Atomics, Version 1.0

If the Ziccamoa extension is implemented, then main memory regions with both the cacheability and

coherence PMAs must support all atomics in the Zaamo extension.

5.5. Ziccamoc Extension for Main Memory Compare-and-Swap, Version 1.0

If the Ziccamoc extension is implemented, then main memory regions with both the cacheability and coherence PMAs must provide **AMOCASQ**-level PMA support.

5.6. Zicclsm Extension for Main Memory Misaligned Accesses, Version 1.0

If the Zicclsm extension is implemented, then misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.



This definition includes vector memory accesses. It does not include instructions in the Zaamo, Zalrsc, Zabha, or Zalasr extensions.



Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

5.7. Zic64b Extension for 64-byte Cache Blocks, Version 1.0

If the Zic64b extension is implemented, then cache blocks must be 64 bytes in size, naturally aligned in the address space.

DRAFT

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.

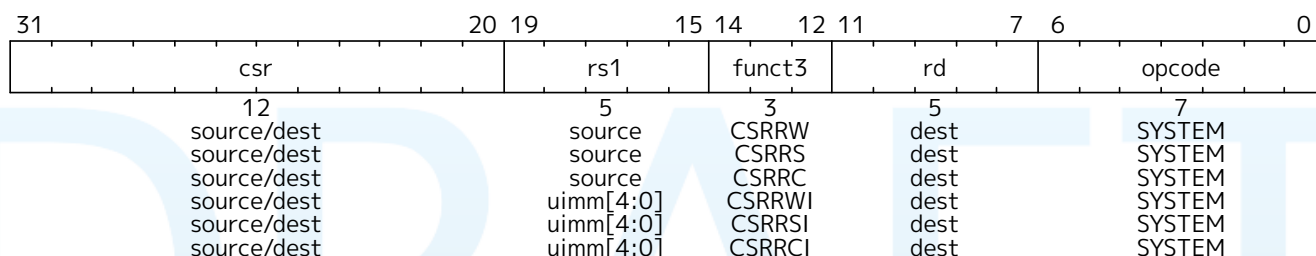


While CSRs are primarily used by the privileged architecture, there are several uses in unprivileged code including for counters and timers, and for floating-point status.

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, CSRWS, 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
CSRWS/CSRRC	-	Yes	Yes	No
CSRWS/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 CSRWS[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, x0`, 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 15](#)), 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 in [Volume II, Section 3.6.5](#). 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 in [Volume II, Section 3.6.5](#). 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 in [Volume II, Section 3.6.5](#). 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 15](#) concerning the RVWMO memory consistency model.

DRAFT

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.

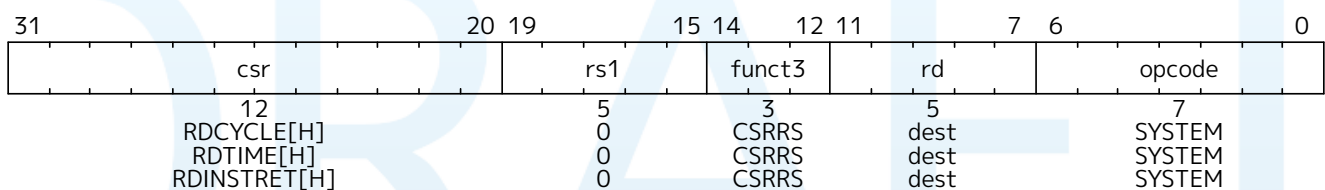
7.1. "Zicntr" Extension for Base Counters and Timers

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".

DRAFT

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, Chapter 5](#).

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.

DRAFT

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.

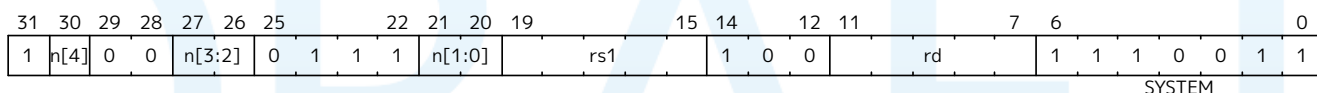


It is sometimes desirable to define instruction-set extensions whose instructions, rather than raising illegal-instruction exceptions when the extension is not implemented, take no useful action (beyond writing $x[rd]$). For example, programs with control-flow integrity checks can execute correctly on 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.

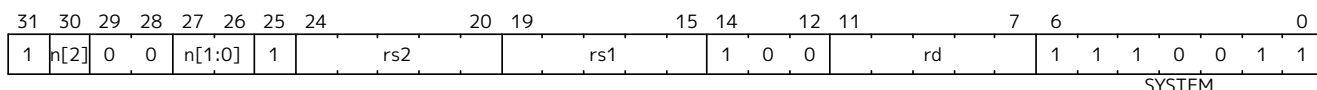
Although similar in some respects to HINTs, MOPs cannot be encoded as HINTs, because unlike HINTs, MOPs are allowed to alter architectural state.

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 *xn*, 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.

15	13	12	11	10	8	7	6	2	1	0
0	1	1	0	0	n[3:1]	1	0	0	0	0



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	<i>x1</i>
C.MOP.3	0110000110000001	<i>x3</i>
C.MOP.5	0110001010000001	<i>x5</i>
C.MOP.7	0110001110000001	<i>x7</i>
C.MOP.9	0110010010000001	<i>x9</i>
C.MOP.11	0110010110000001	<i>x11</i>
C.MOP.13	0110011010000001	<i>x13</i>
C.MOP.15	0110011110000001	<i>x15</i>



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
✓	✓	<code>czero.eqz rd, rs1, rs2</code>	Conditional zero, if condition is equal to zero
✓	✓	<code>czero.nez rd, rs1, rs2</code>	Conditional zero, if condition is nonzero

11.1. Instructions (in alphabetical order)

11.1.1. czero.eqz

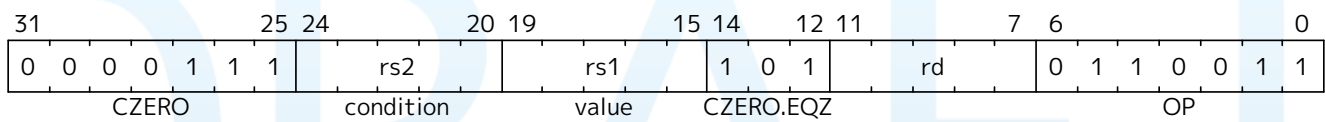
Synopsis

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

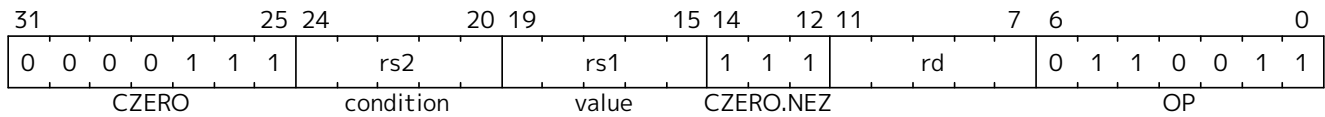
Synopsis

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 $rd = (rc == 0) ? (rs1 + rs2) : rs1$	<code>czero.nez rd, rs2, rc</code> <code>add rd, rs1, rd</code>	2 insns
Conditional add, if non-zero $rd = (rc != 0) ? (rs1 + rs2) : rs1$	<code>czero.eqz rd, rs2, rc</code> <code>add rd, rs1, rd</code>	
Conditional subtract, if zero $rd = (rc == 0) ? (rs1 - rs2) : rs1$	<code>czero.nez rd, rs2, rc</code> <code>sub rd, rs1, rd</code>	
Conditional subtract, if non-zero $rd = (rc != 0) ? (rs1 - rs2) : rs1$	<code>czero.eqz rd, rs2, rc</code> <code>sub rd, rs1, rd</code>	
Conditional bitwise-or, if zero $rd = (rc == 0) ? (rs1 rs2) : rs1$	<code>czero.nez rd, rs2, rc</code> <code>or rd, rs1, rd</code>	
Conditional bitwise-or, if non-zero $rd = (rc != 0) ? (rs1 rs2) : rs1$	<code>czero.eqz rd, rs2, rc</code> <code>or rd, rs1, rd</code>	
Conditional bitwise-xor, if zero $rd = (rc == 0) ? (rs1 ^ rs2) : rs1$	<code>czero.nez rd, rs2, rc</code> <code>xor rd, rs1, rd</code>	
Conditional bitwise-xor, if non-zero $rd = (rc != 0) ? (rs1 ^ rs2) : rs1$	<code>czero.eqz rd, rs2, rc</code> <code>xor rd, rs1, rd</code>	
Conditional bitwise-and, if zero $rd = (rc == 0) ? (rs1 \& rs2) : rs1$	<code>and rd, rs1, rs2</code> <code>czero.eqz rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	3 insns (requires 1 temporary)
Conditional bitwise-and, if non-zero $rd = (rc != 0) ? (rs1 \& rs2) : rs1$	<code>and rd, rs1, rs2</code> <code>czero.nez rtmp, rs1, rc</code> <code>or rd, rd, rtmp</code>	
Conditional select, if zero $rd = (rc == 0) ? rs1 : rs2$	<code>czero.nez rd, rs1, rc</code> <code>czero.eqz rtmp, rs2, rc</code> <code>add rd, rd, rtmp</code>	
Conditional select, if non-zero $rd = (rc != 0) ? rs1 : rs2$	<code>czero.eqz rd, rs1, rc</code> <code>czero.nez rtmp, rs2, rc</code> <code>add rd, rd, rtmp</code>	

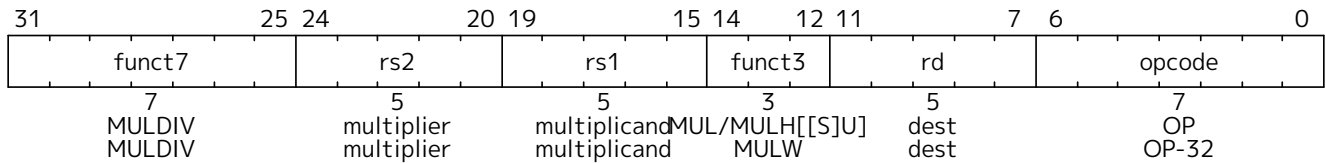
Chapter 12. "M" Extension for Integer Multiplication and Division, Version 2.0

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.



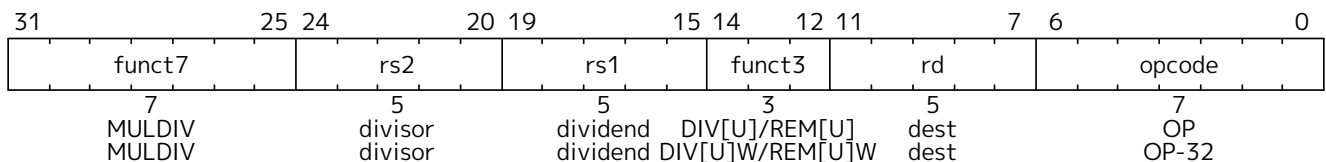
MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the multiplicand (which contains the sign bit) with the less-significant words of the multiplier (which are unsigned).

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 $\text{dividend} = \text{divisor} \times \text{quotient} + \text{remainder}$.

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.

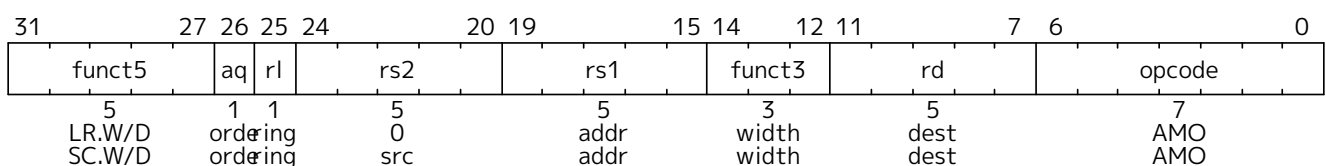
13.1. Specifying Ordering of Atomic Instructions

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 15.1](#).

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 15.1](#) 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 Zlsrc 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

31					27	26	25	24	20					19	15			14	12		11	7			6	0				
funct5					aq	rl	rs2			rs1					funct3			rd			opcode									
5					1	1	5			5					3			5			7									
AMOSWAP.W/D							src			addr					width			dest			AMO									
AMOADD.W/D							src			addr					width			dest			AMO									
AMOAND.W/D							src			addr					width			dest			AMO									
AMOOR.W/D							src			addr					width			dest			AMO									
AMOXOR.W/D							src			addr					width			dest			AMO									
AMOMAX[U].W/D							src			addr					width			dest			AMO									
AMOMIN[U].W/D							src			addr					width			dest			AMO									

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, Section 3.6.4](#), 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 Zlsrc 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. "Za" Atomic Instructions

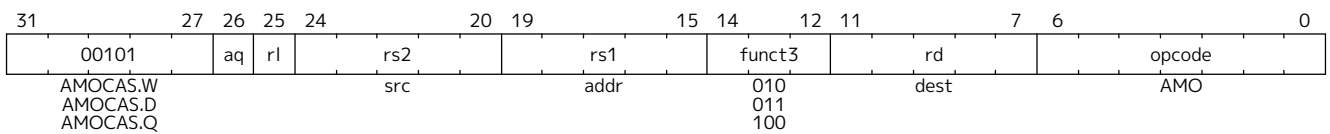


This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

14.1. "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.

14.1.1. 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*

14.2. "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.

14.2.1. 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.

31	27	26	25	24	20	19	15	14	12	11	7	6	0				
funct5				aq	rl	rs2			rs1			funct3		rd		opcode	
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.

14.3. "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.

14.3.1. 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, Section 3.6.4](#), 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.

14.3.2. 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

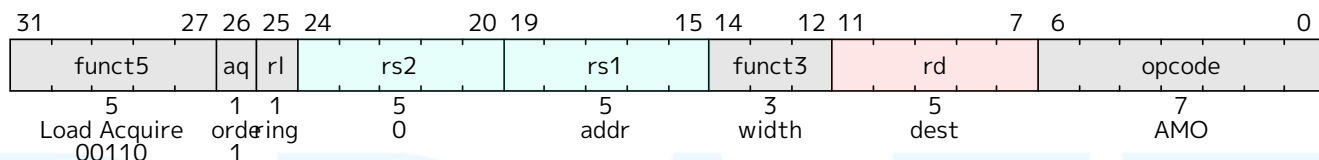
`lb.{aq,aqrl} rd, (rs1)`

`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 ($2^{\text{width}+3}$) 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.*

14.3.3. 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

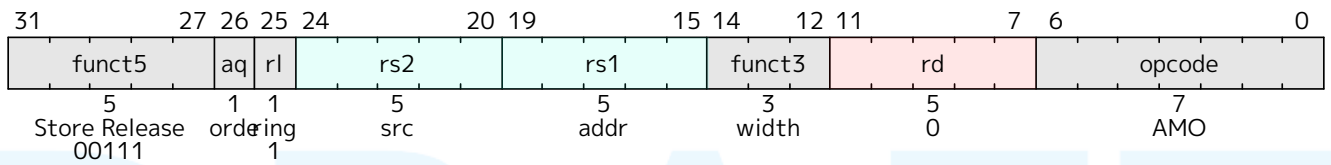
sb.{rl,aqrl} rs2, (rs1)

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 a "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.

14.4. Za128rs Extension for Reservation-Set Size, Version 1.0

The Za128rs extension requires that the reservation sets used by the instructions in the Zalrsc extension be contiguous, naturally aligned, and at most 128 bytes in size.

14.5. Za64rs Extension for Reservation-Set Size, Version 1.0

The Za64rs extension requires that the reservation sets used by the instructions in the Zalrsc extension be contiguous, naturally aligned, and at most 64 bytes in size.

The Za64rs extension implies the Za128rs extension.

14.6. "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.

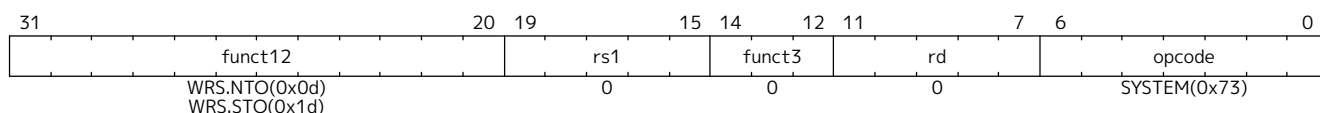
Sometimes the program waiting on a memory update may also need to carry out a task at a future time or otherwise place an upper bound on the wait. To support such use cases a second instruction **WRS.STO** (WRS-with-short-timeout) is provided that works like **WRS.NTO** but bounds the stall duration to an implementation-defined 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.

14.6.1. 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.*

14.7. Zama16b Extension for 16-byte Misaligned Atomicity, Version 1.0

If the Zama16b extension is implemented, then the [misaligned atomicity granule](#) in main memory regions with both the cacheability and coherence PMAs is 16 bytes. Misaligned loads, stores, and AMOs to main memory regions that do not cross a naturally aligned 16-byte boundary are atomic.

Chapter 15. 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 16](#)) 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.

15.1. 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).

15.1.1. 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 16](#)). 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.

15.1.2. 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 15.2](#).

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 15.3](#) 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 15.3](#)
- 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 15.3](#)
- 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 15.3](#).

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.

15.1.3. 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

15.1.4. 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*.

15.1.4.1. 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

15.1.4.2. 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.

15.1.4.3. Progress Axiom

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

15.2. CSR Dependency Tracking Granularity

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

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

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

15.3. 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 15.1.2](#).

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:

- ^AAddress source register
- ^DData 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		<i>rd</i>		
AUIPC		<i>rd</i>		
JAL		<i>rd</i>		
JALR [†]	<i>rs1</i>	<i>rd</i>		
BEQ	<i>rs1, rs2</i>			
BNE	<i>rs1, rs2</i>			
BLT	<i>rs1, rs2</i>			
BGE	<i>rs1, rs2</i>			
BLTU	<i>rs1, rs2</i>			
BGEU	<i>rs1, rs2</i>			
LB [†]	<i>rs1</i> ^A	<i>rd</i>		
LH [†]	<i>rs1</i> ^A	<i>rd</i>		
LW [†]	<i>rs1</i> ^A	<i>rd</i>		
LBU [†]	<i>rs1</i> ^A	<i>rd</i>		
LHU [†]	<i>rs1</i> ^A	<i>rd</i>		
SB	<i>rs1</i> ^A , <i>rs2</i> ^D			
SH	<i>rs1</i> ^A , <i>rs2</i> ^D			
SW	<i>rs1</i> ^A , <i>rs2</i> ^D			
ADDI	<i>rs1</i>	<i>rd</i>		
SLTI	<i>rs1</i>	<i>rd</i>		
SLTIU	<i>rs1</i>	<i>rd</i>		
XORI	<i>rs1</i>	<i>rd</i>		
ORI	<i>rs1</i>	<i>rd</i>		
ANDI	<i>rs1</i>	<i>rd</i>		
SLLI	<i>rs1</i>	<i>rd</i>		
SRLI	<i>rs1</i>	<i>rd</i>		
SRAI	<i>rs1</i>	<i>rd</i>		

	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	<i>rs1, rs2</i>	<i>rd</i>		
SRAW	<i>rs1, rs2</i>	<i>rd</i>		

Table 15. RV32M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
MUL	<i>rs1, rs2</i>	<i>rd</i>		
MULH	<i>rs1, rs2</i>	<i>rd</i>		
MULHSU	<i>rs1, rs2</i>	<i>rd</i>		
MULHU	<i>rs1, rs2</i>	<i>rd</i>		
DIV	<i>rs1, rs2</i>	<i>rd</i>		
DIVU	<i>rs1, rs2</i>	<i>rd</i>		
REM	<i>rs1, rs2</i>	<i>rd</i>		
REMU	<i>rs1, rs2</i>	<i>rd</i>		

Table 16. RV64M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
MULW	<i>rs1, rs2</i>	<i>rd</i>		
DIVW	<i>rs1, rs2</i>	<i>rd</i>		
DIVUW	<i>rs1, rs2</i>	<i>rd</i>		
REMW	<i>rs1, rs2</i>	<i>rd</i>		
REMUW	<i>rs1, rs2</i>	<i>rd</i>		

Table 17. RV32A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.W [†]	<i>rs1</i> ^A	<i>rd</i>		
SC.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i> [*]		* if successful
AMOSWAP.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOADD.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOXOR.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOAND.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOOR.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOMIN.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOMAX.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOMINU.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOMAXU.W [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		

Table 18. RV64A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.D [†]	<i>rs1</i> ^A	<i>rd</i>		
SC.D [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i> [*]		* if successful
AMOSWAP.D [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		

	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	<i>rs1</i> , <i>frm</i> *	<i>rd</i>	NX	*if <i>rm</i> =111
FMV.D.X	<i>rs1</i>	<i>rd</i>		

DRAFT

Chapter 16. "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 15.1](#).



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 15.1.4.1](#) 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 17. "CMO" Extensions for Base Cache Management Operation ISA, Version 1.0.0

17.1. Pseudocode for instruction semantics

The semantics of each instruction in the [Instructions](#) chapter is expressed in a SAIL-like syntax.

17.2. 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 chapter 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: **CB0.INVALID**, **CB0.CLEAN**, and **CB0.FLUSH**
- The *Zicboz* extension defines a cache-block zero instruction: **CB0.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 chapter 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.

17.3. Background

This chapter provides information common to all CMO extensions.

17.3.1. 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](#).

17.3.2. 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](#).

17.4. 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*.

17.4.1. Memory Ordering

17.4.1.1. 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.

17.4.1.2. 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.

17.4.2. 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.

17.4.2.1. 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.

17.4.2.2. 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.

17.4.2.3. Address-Misaligned Exceptions

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

17.4.2.4. 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`

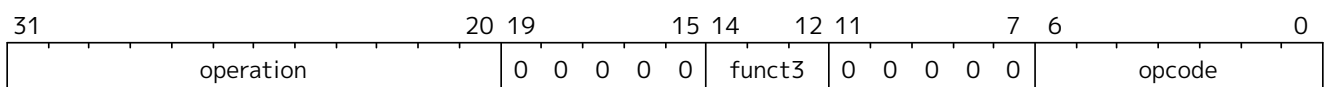
If the Zicbop extension is implemented, this specification recommends the following additional trigger module behaviors:

- 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.

17.4.2.5. 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.

17.4.3. 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.

17.4.4. 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.

17.5. 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.INVALID` 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 CB0.INVALID and perform flush operation>
    }
    else
    {
        <execute CB0.INVALID and perform invalidate operation>
    }
}
}

```

Until a modified cache block has updated memory, a **CB0.INVALID** 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 **CB0.INVALID** either traps or performs a flush operation in a lower privileged level.

A **CB0.CLEAN** or **CB0.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 CB0.CLEAN or CB0.FLUSH>
}

```

Finally, a **CB0.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) && !henvcfg.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 CB0.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.

17.6. Extensions

CMO instructions are defined in the following extensions:

- [Cache-Block Management Instructions](#)
- [Cache-Block Zero Instructions](#)
- [Cache-Block Prefetch Instructions](#)

17.6.1. 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 <i>base</i>	Cache Block Clean
✓	✓	cbo.flush <i>base</i>	Cache Block Flush
✓	✓	cbo.inval <i>base</i>	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.

17.6.2. 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 <i>base</i>	Cache Block Zero

17.6.3. 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 **0b000000**. 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 0b000000; 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 <i>offset(base)</i>	Cache Block Prefetch for Instruction Fetch
✓	✓	prefetch.r <i>offset(base)</i>	Cache Block Prefetch for Data Read
✓	✓	prefetch.w <i>offset(base)</i>	Cache Block Prefetch for Data Write

17.7. Instructions

17.7.1. cbo.clean

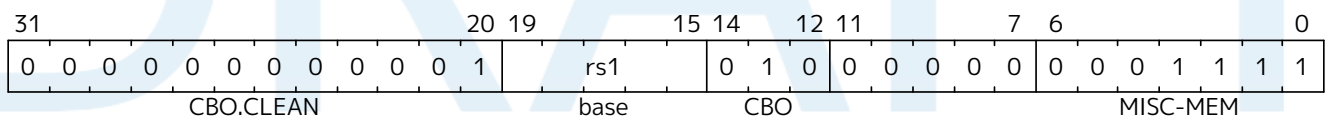
Synopsis

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 **cho.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.

17.7.2. cbo.flush

Synopsis

Perform a flush operation on a cache block

Mnemonic

$$\text{cbo.flush offset}(base)$$

Encoding

31	20	19	15	14	12	11	7	6	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0
CBO.FLUSH									
rs1									
base									
CBO									
MISC-MEM									

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.

17.7.3. cbo.inval

Synopsis

Perform an invalidate operation on a cache block

Mnemonic

$$\text{cbo.inval } \textit{offset}(\textit{base})$$

Encoding

31										20 19				15 14				12 11				7 6				0	
0	0	0	0	0	0	0	0	0	0	rs1				0	1	0	0	0	0	0	0	0	1	1	1	1	
CBO.INVAL										base				CBO				MISC-MEM									

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.

17.7.4. cbo.zero

Synopsis

Store zeros to the full set of bytes corresponding to a cache block

Mnemonic

$$\text{cbo.zero offset}(base)$$

Encoding

The RISC-V Instruction Set Manual | © RISC-V International

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 **0b000000**, 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.

17.7.7. prefetch.w

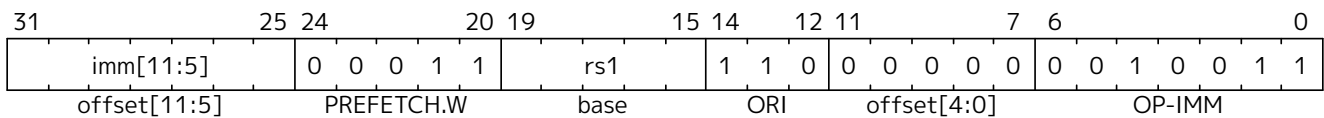
Synopsis

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 **0b000000**, 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 18. "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.

18.1. 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 **fcscr**, 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	

18.2. 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](#).

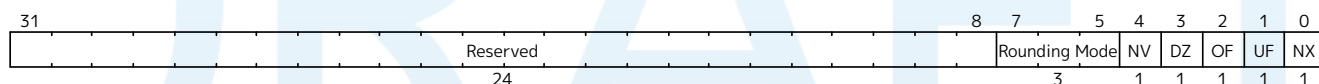


Figure 2. Floating-point control and status register

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 $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		<i>Reserved for future use.</i>
110		<i>Reserved for future use.</i>
111	DYN	In instruction's <i>rm</i> field, selects dynamic rounding mode; In Rounding Mode register, <i>reserved</i> .

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.

18.3. 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.

18.4. Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

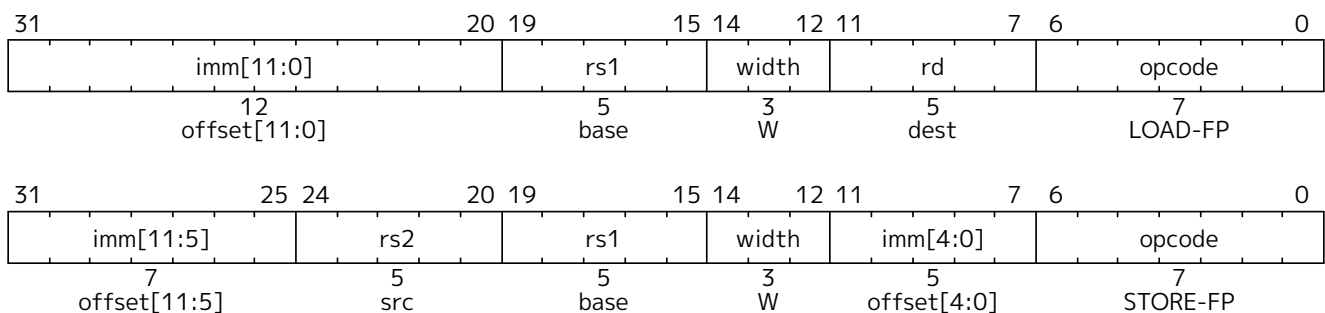
In the parlance of the IEEE standard, tininess is detected after rounding.



Detecting tininess after rounding results in fewer spurious underflow signals.

18.5. 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.

18.6. 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

<i>fmt</i> 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.*

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5					rs2		rs1		rm		rd		opcode
5					5		5		3		5		7
FADD/FSUB					src2		src1		RM		dest		OP-FP
FMUL/FDIV					src2		src1		RM		dest		OP-FP
FSQRT					0		src				dest		OP-FP
FMIN-MAX					src2		src1		MIN/MAX		dest		OP-FP

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 $(rs1 \times rs2) + rs3$.

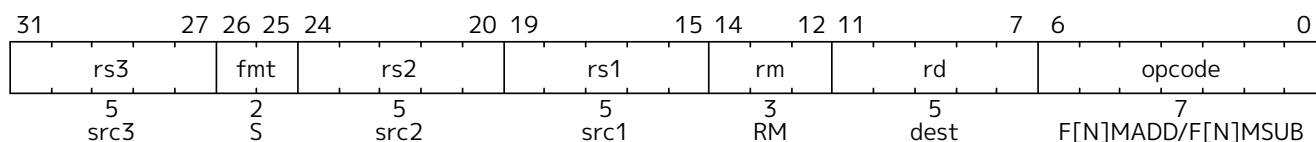
FMSUB.S multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. FMSUB.S computes $(rs1 \times rs2) - rs3$.

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 $-(rs1 \times rs2) + rs3$.

FNMAADD.S multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. FNMAADD.S computes $-(rs1 \times rs2) - rs3$.



The FNMSUB and FNMAADD 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 FNMAADD 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 $\infty \times 0 + qNaN$.

18.7. 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.W.U.S, FCVT.L.U.S, FCVT.S.W.U, and FCVT.S.L.U 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.

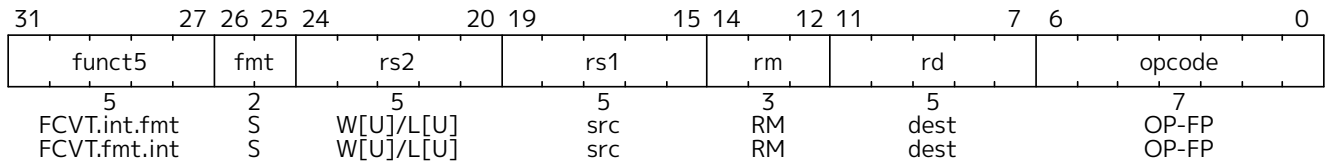
All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using FCVT.S.W *rd*, **x0**, which will never set any exception flags.

Table 27. Domains of float-to-integer conversions and behavior for invalid inputs

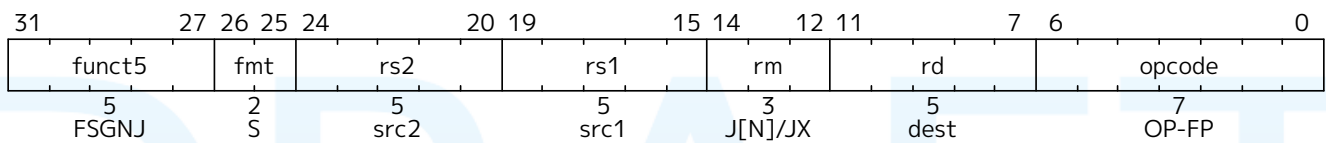
	FCVT.W.S	FCVT.W.U.S	FCVT.L.S	FCVT.L.U.S
Minimum valid input (after rounding)	-2^{31}	0	-2^{63}	0
Maximum valid input (after rounding)	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$
Output for out-of-range negative input	-2^{31}	0	-2^{63}	0
Output for $-\infty$	-2^{31}	0	-2^{63}	0

	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*).

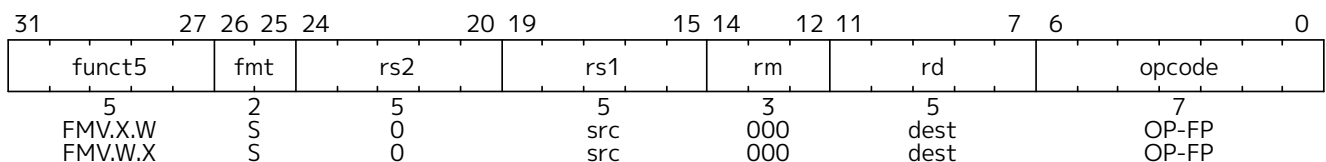


The sign-injection instructions provide floating-point MV, ABS, and NEG, as well as supporting a few other operations, including the IEEE copySign operation and sign manipulation in transcendental math function libraries. Although MV, ABS, and NEG only need a single register operand, whereas FSGNJ instructions 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.



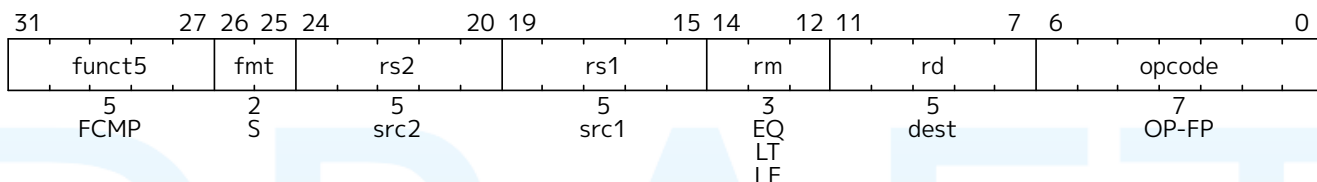


The base floating-point ISA was defined so as to allow implementations to employ an internal recoding of the floating-point format in registers to simplify handling of subnormal values and possibly to reduce functional unit latency. To this end, the F extension avoids representing integer values in the 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.

18.8. Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers ($rs1 = rs2$, $rs1 < rs2$, $rs1 \leq rs2$) 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 $\leq$ comparison, whereas the base ISAs provide a $\geq$ branch comparison. Because $\leq$ can be synthesized from $\geq$ and vice-versa, there is no performance implication to this inconsistency, but it is nevertheless an unfortunate incongruity in the ISA.

18.9. 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.

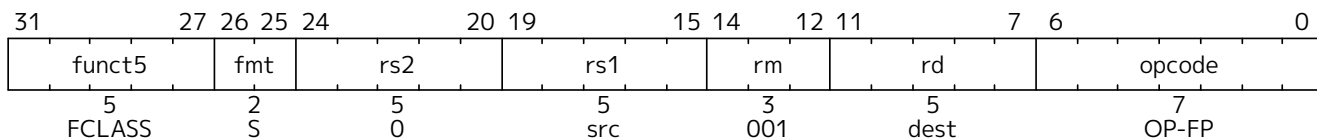


Table 28. Format of result of FCLASS instruction.

rd bit	Meaning
0	$rs1$ is $-\infty$.
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.

<i>rd</i> bit	Meaning
6	<i>rs1</i> is a positive normal number.
7	<i>rs1</i> is $+\infty$.
8	<i>rs1</i> is a signaling NaN.
9	<i>rs1</i> is a quiet NaN.

DRAFT

Chapter 19. "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.

19.1. 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 19.2](#).



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.

19.2. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower n -bit types, $n < \text{FLEN}$, 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, $n < m \leq \text{FLEN}$. 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-boxed value.



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, $n < \text{FLEN}$, 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, $n < \text{FLEN}$, 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 extension 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.

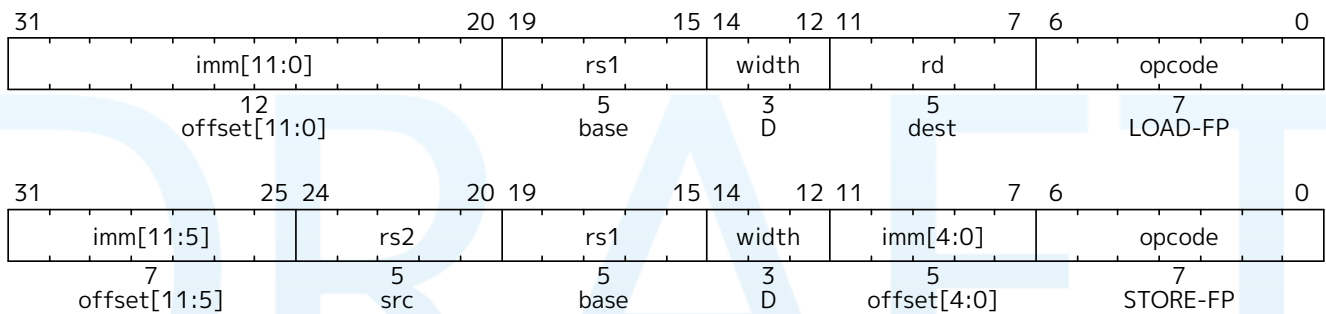
Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can 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.

19.3. 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.

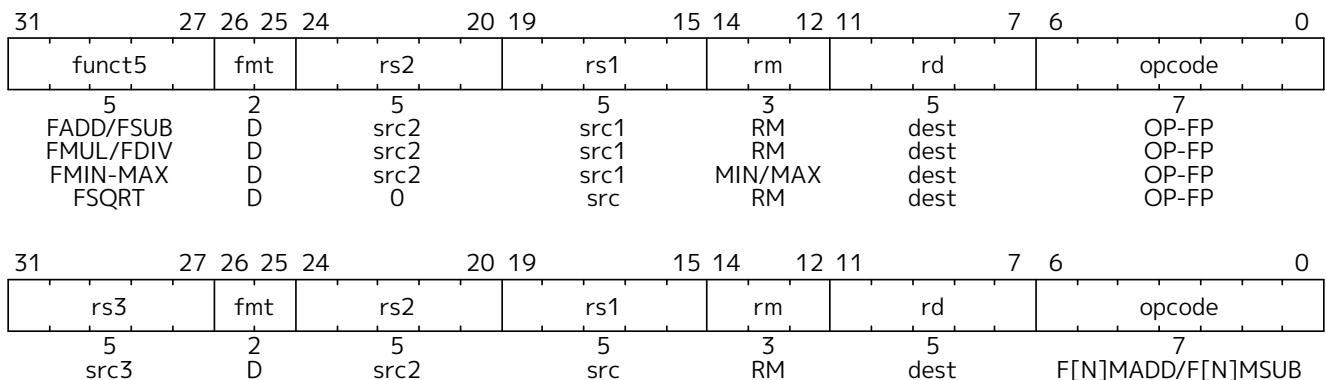


FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and $XLEN \geq 64$.

FLD and FSD do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

19.4. Double-Precision Floating-Point Computational Instructions

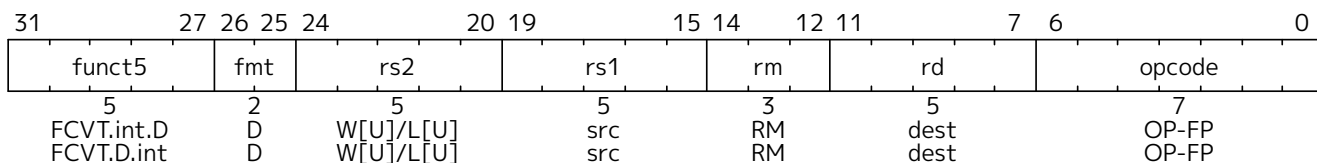
The double-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands and produce double-precision results.



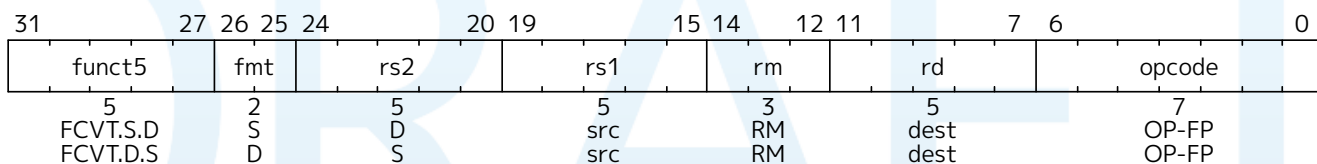
19.5. Double-Precision Floating-Point Conversion and Move Instructions

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.W.U.D, FCVT.L.U.D, FCVT.D.W.U, and FCVT.D.L.U 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.

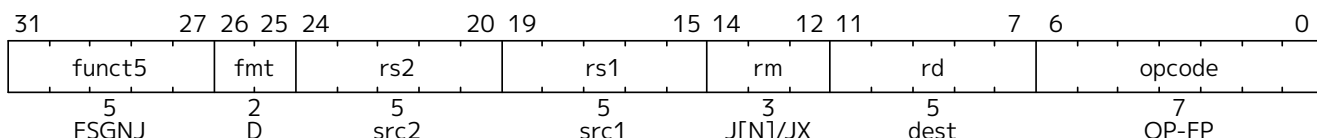
All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. Note FCVT.D.W[U] always produces an exact result and is unaffected by rounding mode.



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.

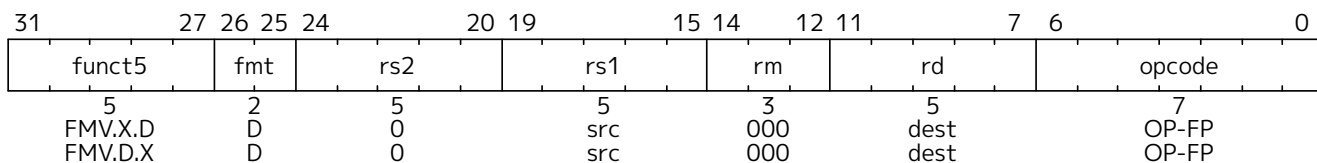


Floating-point to floating-point sign-injection instructions, FSGNJ.D, FSGNJN.D, and FSGNJX.D are defined analogously to the single-precision sign-injection instruction.



For $XLEN \geq 64$ 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.



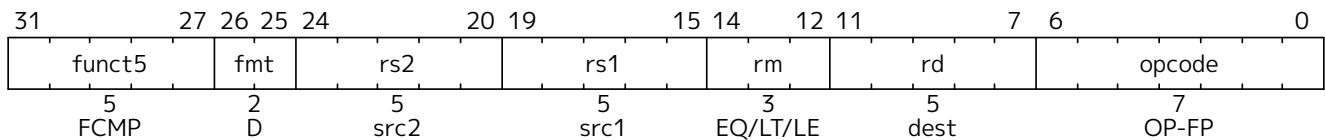
Early versions of the RISC-V ISA had additional instructions to allow RV32 systems to transfer between the upper and lower portions of a 64-bit floating-point register and an integer register. However, these would be the only instructions with partial register writes and would

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.

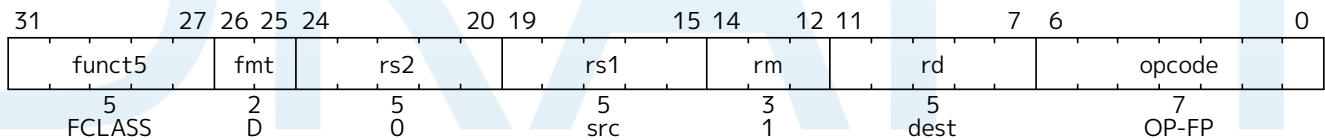
19.6. Double-Precision Floating-Point Compare Instructions

The double-precision floating-point compare instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands.



19.7. 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.

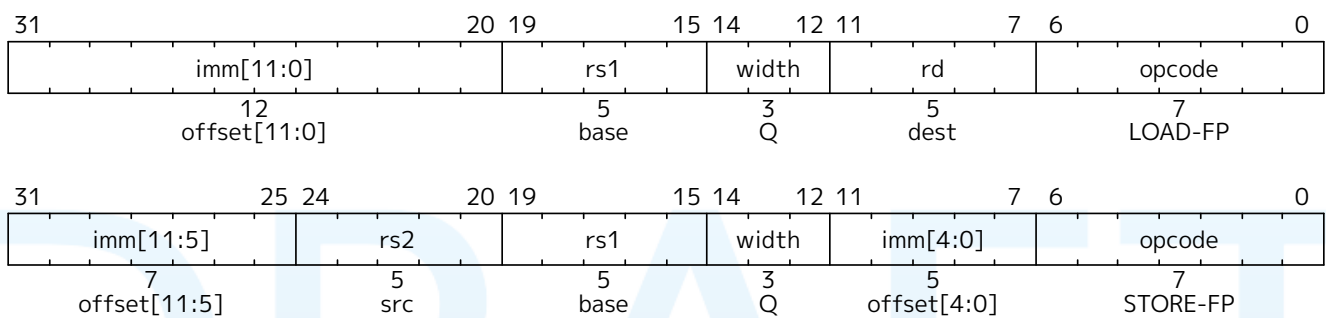


Chapter 20. "Q" Extension for Quad-Precision Floating-Point, Version 2.2

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 19.2](#) 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.

20.1. 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.

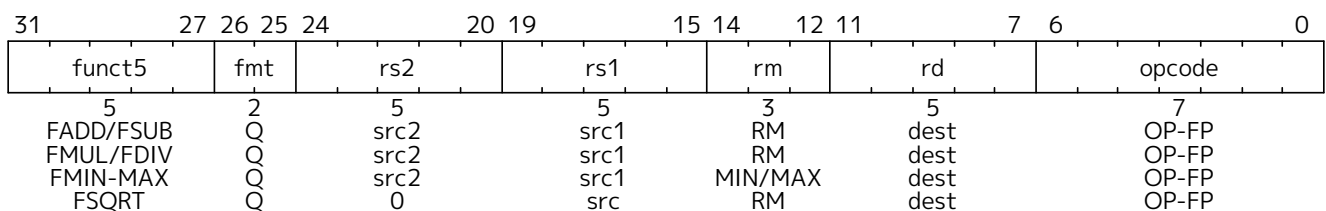
20.2. Quad-Precision Computational Instructions

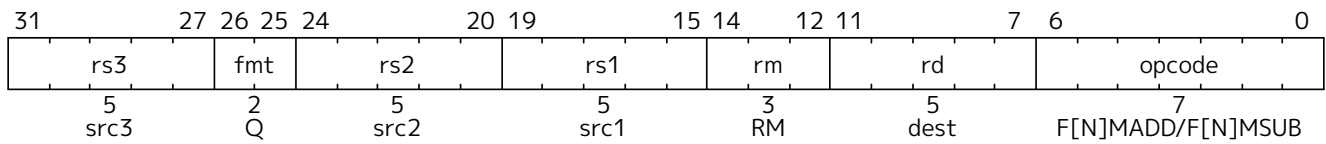
A new supported format is added to the format field of most instructions, as shown in [Table 29](#)

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

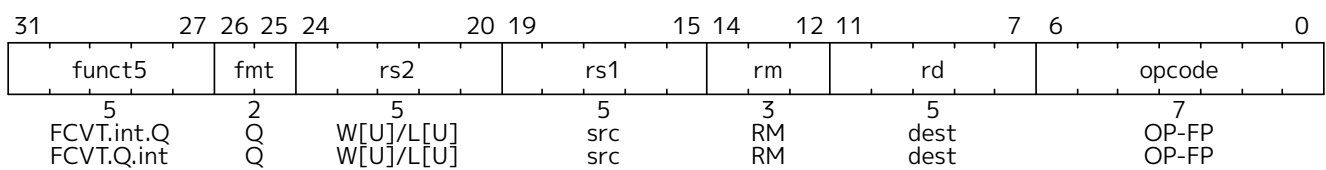
The quad-precision floating-point computational instructions are defined analogously to their double-precision counterparts, but operate on quad-precision operands and produce quad-precision results.



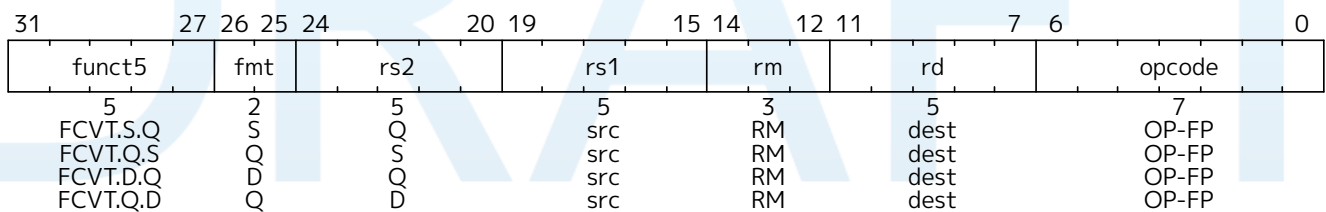


20.3. Quad-Precision Convert and Move Instructions

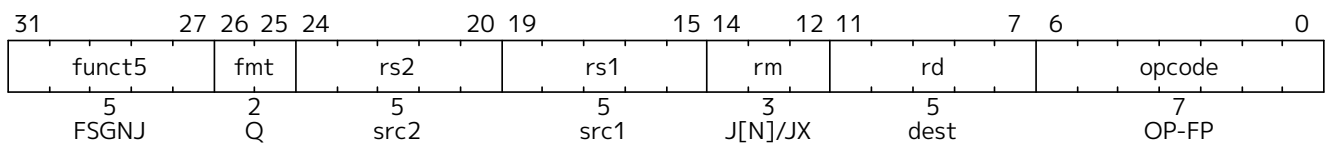
New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision-to-integer and integer-to-double-precision conversion instructions. FCVT.W.Q or FCVT.L.Q converts a quad-precision floating-point number to 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.



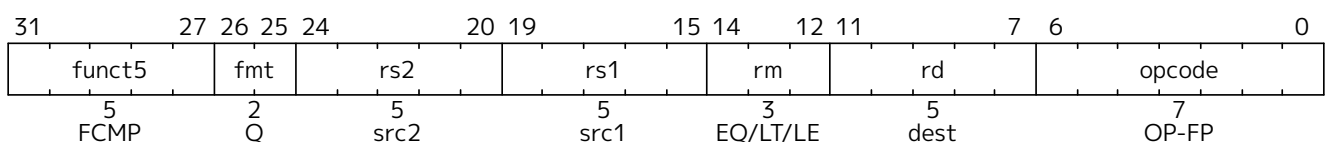
Floating-point to floating-point sign-injection instructions, FSGNJ.Q, FSGNJN.Q, and FSGNJX.Q are defined analogously to the double-precision sign-injection instruction.



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.

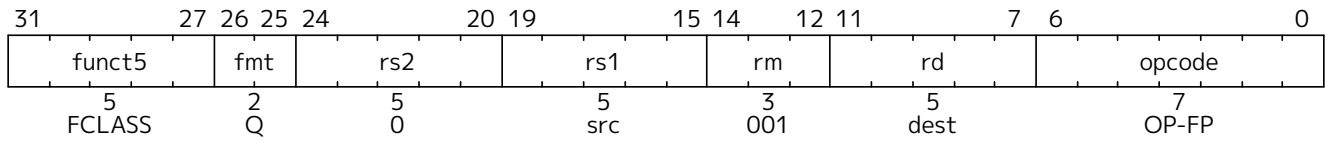
20.4. Quad-Precision Floating-Point Compare Instructions

The quad-precision floating-point compare instructions are defined analogously to their double-precision counterparts, but operate on quad-precision operands.



20.5. 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.



DRAFT

Chapter 21. "Zfh" and "Zfhmin" Extensions for Half-Precision Floating-Point, Version 1.0

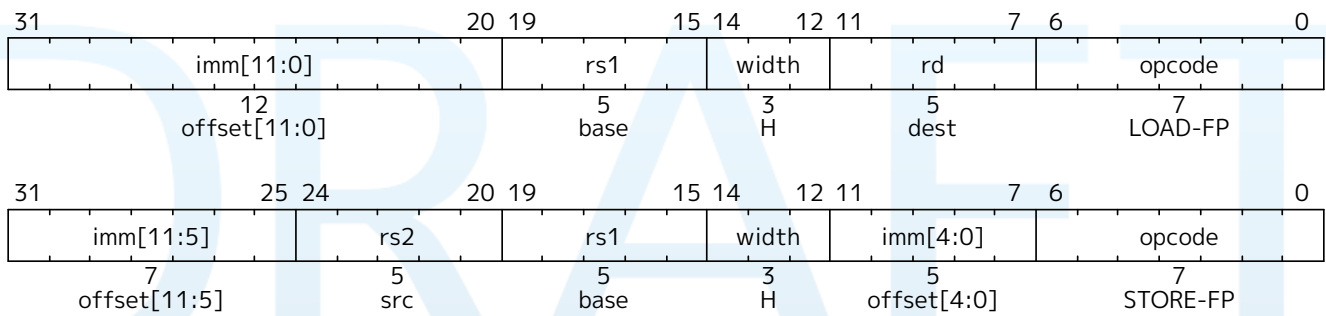
This chapter describes the Zfh standard extension for 16-bit half-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The Zfh extension depends on the single-precision floating-point extension, F. The NaN-boxing scheme described in [Section 19.2](#) is extended 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., $\text{half} \times \text{half} + \text{single} \rightarrow \text{single}$ —or implicitly narrow their results—e.g., $\text{half} + \text{single} \rightarrow \text{half}$.

21.1. 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*.

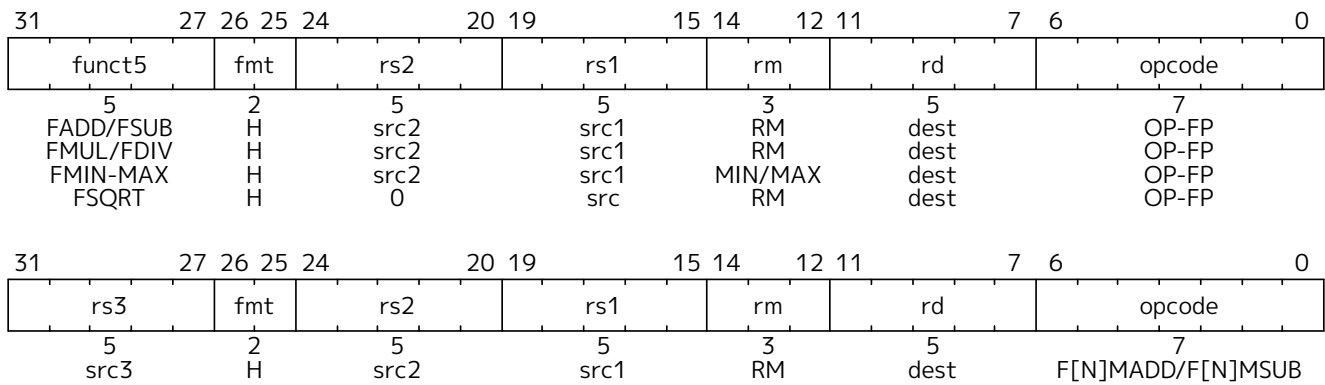
21.2. Half-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in [Table 30](#).

Table 30. Format field encoding.

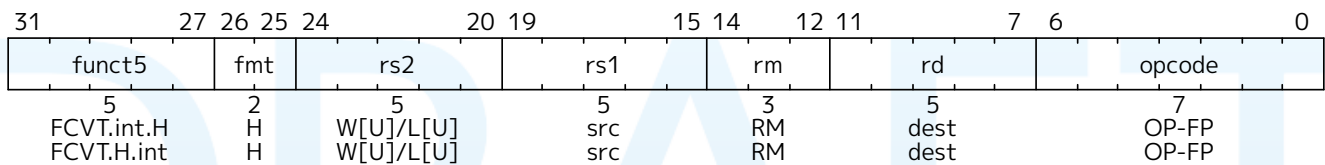
fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

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.

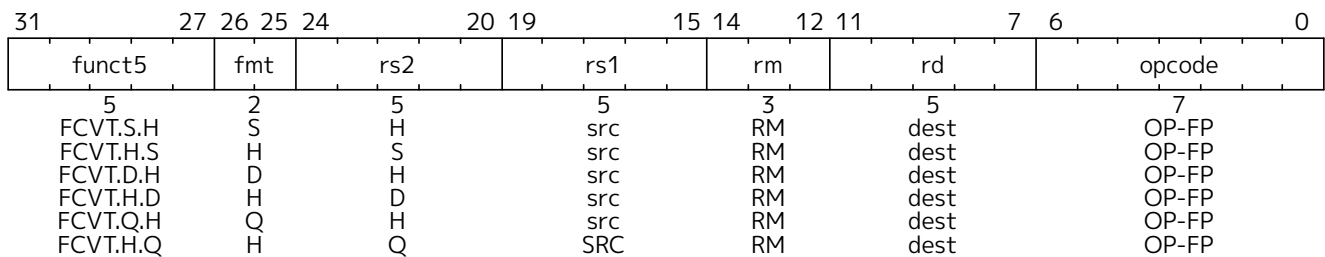


21.3. Half-Precision Conversion and Move Instructions

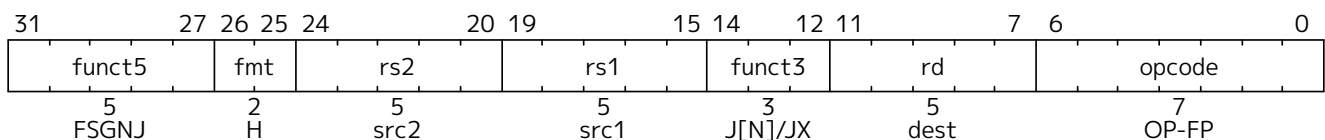
New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the single-precision-to-integer and integer-to-single-precision conversion instructions. FCVT.W.H or FCVT.L.H converts a half-precision floating-point number to 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.



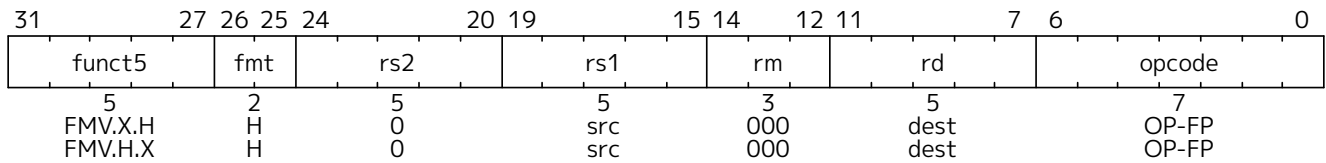
Floating-point to floating-point sign-injection instructions, FSGNJ.H, FSGNJN.H, and FSGNJX.H are defined analogously to the single-precision sign-injection instruction.



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.

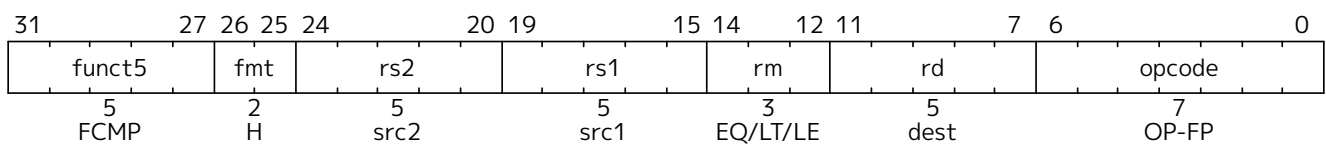
FMV.H.X moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register *rs1* to the floating-point register *rd*, NaN-boxing the result.

FMV.X.H and FMV.H.X do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.



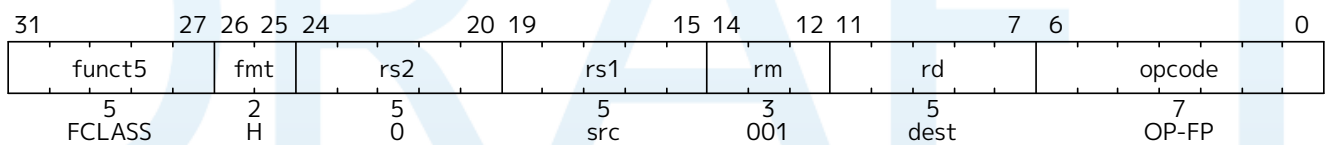
21.4. Half-Precision Floating-Point Compare Instructions

The half-precision floating-point compare instructions are defined analogously to their single-precision counterparts, but operate on half-precision operands.



21.5. 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.



21.6. "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.

DRAFT

Chapter 22. "BF16" Extensions for BFloat16-precision Floating-Point, Version 1.0

22.1. 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.

22.2. 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.

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

22.3.2. 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.

22.3.2.1. 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.

22.3.2.2. 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.

22.3.2.3. 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.

22.3.2.4. 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.

22.3.2.5. 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.

22.3.2.6. 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.

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.

DRAFT

22.4. 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.

22.4.1. 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 21](#).



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	Section 22.5.1
FCVT.S.BF16	Section 22.5.2
FLH	
FSH	
FMV.H.X	
FMV.X.H	

22.4.2. 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	Section 22.5.3
vfwcvtbf16.f.f.v	Section 22.5.4

22.4.3. Zvfbfmma - 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	Section 22.5.5

22.5. Instructions

22.5.1. fcvt.bf16.s

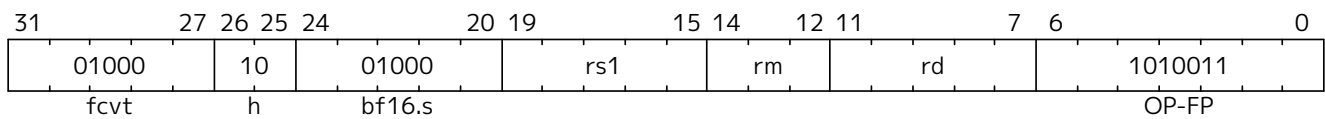
Synopsis

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: [Section 22.4.1](#)

22.5.2. fcvt.s.bf16

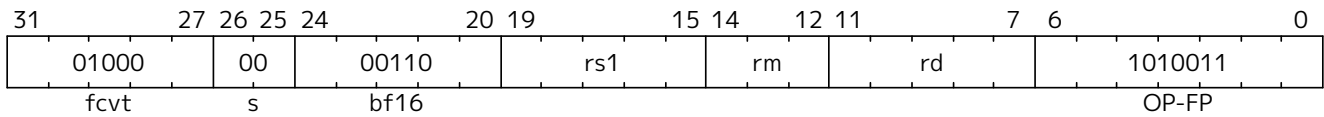
Synopsis

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: [Section 22.4.1](#)

22.5.3. vfnvbtbf16.f.f.w

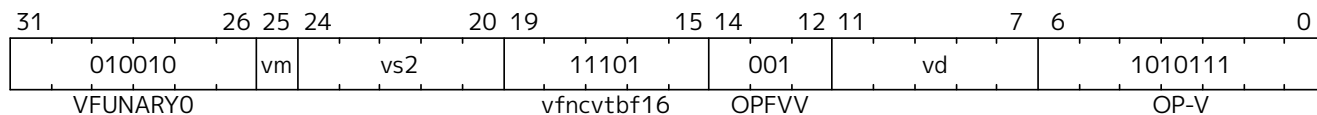
Synopsis

Vector convert FP32 to BF16

Mnemonic

vfnvbtbf16.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 **vfnvbt.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: [Section 22.4.2](#)

22.5.4. vfwcvtbf16.f.f.v

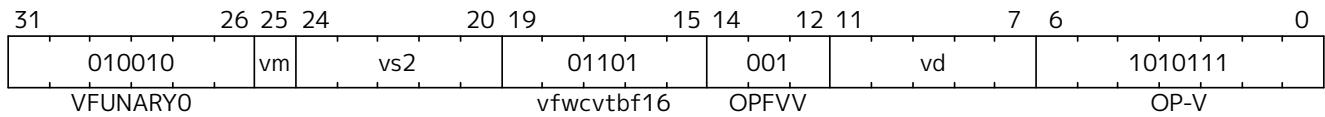
Synopsis

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: [Section 22.4.2](#)

22.5.5. vfwmaccbf16

Synopsis

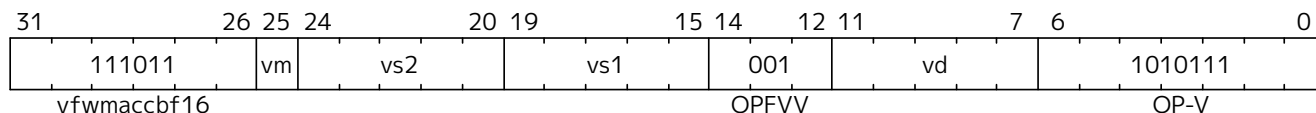
Vector BF16 widening multiply-accumulate

Mnemonic

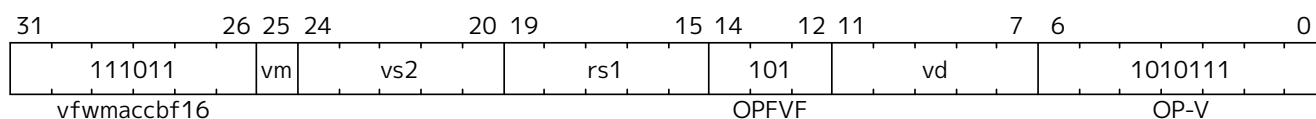
vfwmaccbf16.vv vd, vs1, vs2, vm

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: [Section 22.4.3](#)

Bibliography

[754-2019 - IEEE Standard for Floating-Point Arithmetic](#)

[754-2008 - IEEE Standard for Floating-Point Arithmetic](#)

DRAFT

Chapter 23. "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.

23.1. 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.

<i>rs1</i>	Value	Sign	Exponent	Significand
0	-1.0	1	01111111	000...000
1	<i>Minimum positive normal</i>	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

<i>rs1</i>	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.

23.2. 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.

23.3. 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.

23.4. 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., `fcvtnmod.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} .

23.5. 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.

23.6. 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 24. "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.

24.1. Processing of Narrower Values

Floating-point operands of width $w < \text{XLEN}$ bits occupy bits $w-1:0$ of an **x** register. Floating-point operations on w -bit operands ignore operand bits $\text{XLEN}-1:w$.

Floating-point operations that produce $w < \text{XLEN}$ -bit results fill bits $\text{XLEN}-1:w$ 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.*

24.2. 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.

24.3. 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.

24.4. 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.

24.5. 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.

24.6. Privileged Architecture Implications

As described in [Volume II, Section 3.1.6](#), 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 **mis** 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.

DRAFT

Chapter 25. "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.

25.1. 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)

25.2. 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

25.3. 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: $\times 4$ for words, $\times 8$ for double words, and $\times 16$ 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.

25.3.1. Stack-Pointer-Based Loads and Stores

15	13	12	11	7	6	2	1	0
funct3			imm	rd		imm		op
3	1		5		5		2	
C.LWSP	offset[5]		dest $\neq$ 0		offset[4:2 7:6]		C2	
C.LDSP	offset[5]		dest $\neq$ 0		offset[4:3 8:6]		C2	
C.FLWSP	offset[5]		dest		offset[4:2 7:6]		C2	
C.FLDSP	offset[5]		dest		offset[4:3 8:6]		C2	

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 $rd \neq x0$; 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 $rd \neq x0$; 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)**.

15	13	12	7	6	2	1	0
funct3			imm		rs2		op
3			6		5		2
C.SWSP			offset[5:2 7:6]		src		C2
C.SDSP			offset[5:3 8:6]		src		C2
C.FSWSP			offset[5:2 7:6]		src		C2
C.FDSP			offset[5:3 8:6]		src		C2

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.FDSP 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.

25.3.2. Register-Based Loads and Stores

15	13	12	10	9	7	6	5	4	2	1	0	
funct3			imm			rs1'		imm		rd'		op
3			3			3		2		3		2
C.LW			offset[5:3]			base		offset[2:6]		dest		C0
C.LD			offset[5:3]			base		offset[7:6]		dest		C0
C.FLW			offset[5:3]			base		offset[2:6]		dest		C0
C.FLD			offset[5:3]			base		offset[7:6]		dest		C0

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'*)**.

15			13		12		10			9		7		6		5		4		2		1		0											
funct3						imm						rs1'						imm						rs2'						op					
3						3						3						2						3						2					
C.SW						offset[5:3]						base						offset[2 6]						src						C0					
C.SD						offset[5:3]						base						offset[7:6]						src						C0					
C.FSW						offset[5:3]						base						offset[2 6]						src						C0					
C.FSD						offset[5:3]						base						offset[7:6]						src						C0					

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'*)**.

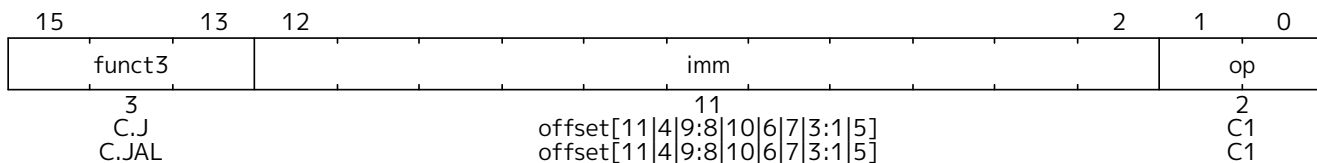
C.SD is an RV64C-only instruction that stores a 64-bit value in register *rs2'* to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register *rs1'*. It expands to **sd *rs2'*, offset(*rs1'*)**.

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')**.

25.4. Control Transfer Instructions

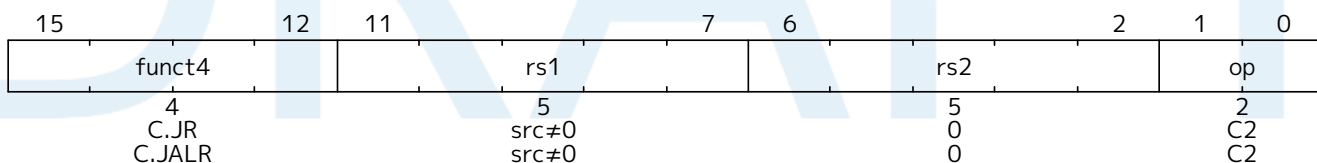
RVC provides unconditional jump instructions and conditional branch instructions. As with base RVI instructions, the offsets of all RVC control transfer instructions are in multiples of 2 bytes.



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**.

CJAL is an RV32C-only instruction that performs the same operation as CJ, but additionally writes the address of the instruction following the jump (**pc+2**) to the link register, **x1**. CJAL expands to **jal x1, offset**.



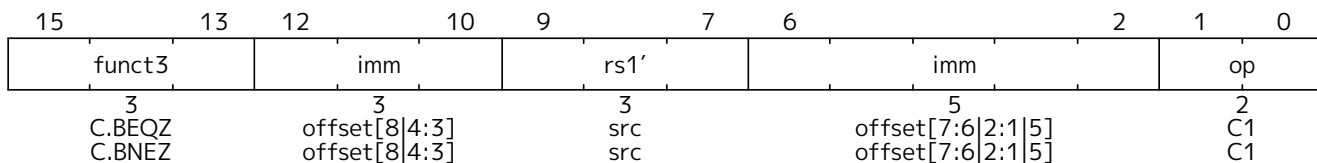
These instructions use the CR format.

CJR (jump register) performs an unconditional control transfer to the address in register *rs1*. CJR expands to `jalr x0, 0(rs1)`. CJR is valid only when *rs1*≠**x0**; the code point with *rs1*=**x0** is reserved.

CJALR (jump and link register) performs the same operation as CJR, but additionally writes the address of the instruction following the jump ($pc+2$) to the link register, `x1`. CJALR expands to `jalr x1, 0(rs1)`. CJALR is valid only when $rs1 \neq x0$; 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**.

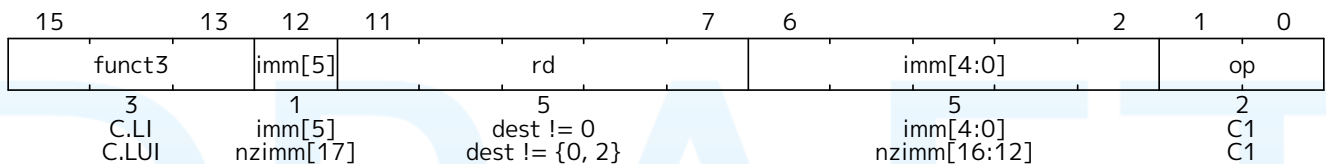
C.BNEZ is defined analogously, but it takes the branch if *rs1'* contains a nonzero value. It expands to **bne rs1', x0, offset**.

25.5. Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

25.5.1. Integer Constant-Generation Instructions

The two constant-generation instructions both use the CI instruction format and can target any integer register.

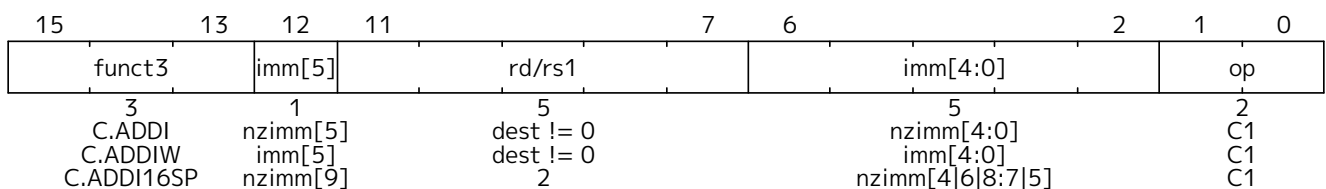


C.LI loads the sign-extended 6-bit immediate, *imm*, into register *rd*. C.LI expands into **addi rd, x0, imm**. The C.LI code points with *rd*=**x0** are HINTs.

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.

25.5.2. 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.



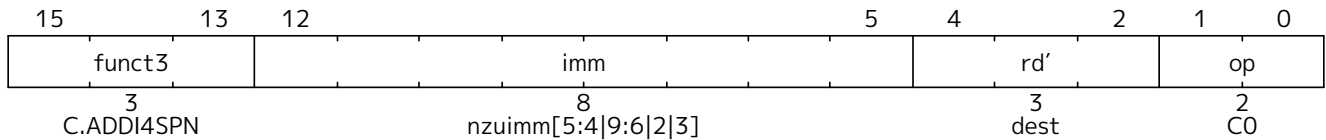
C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register *rd* then writes the result to *rd*. C.ADDI expands into **addi rd, rd, imm**. The code points with *rd*≠0 and *imm*=0 are HINTs. The code points with *rd*=**x0** encode the C.NOP instruction, of which the code points with *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 $rd \neq x0$; the code points with $rd = x0$ are reserved.

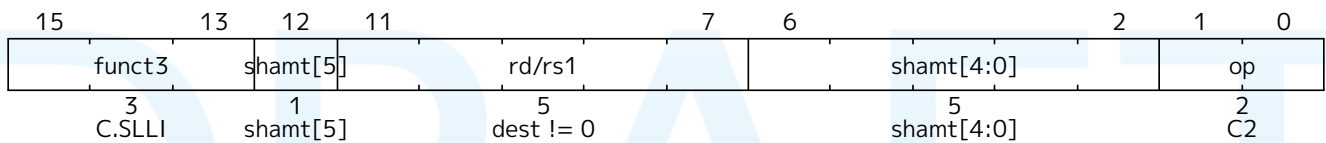
C.ADDI16SP (add immediate to stack pointer) shares the opcode with C.LUI, but has a destination field of `x2`. C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (`sp=x2`), where the immediate is scaled to represent multiples of 16 in the range $[-512, 496]$. 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 $nzimm \neq 0$; the code point with $nzimm = 0$ is reserved.



In the standard RISC-V calling convention, the stack pointer `sp` is always 16-byte aligned.



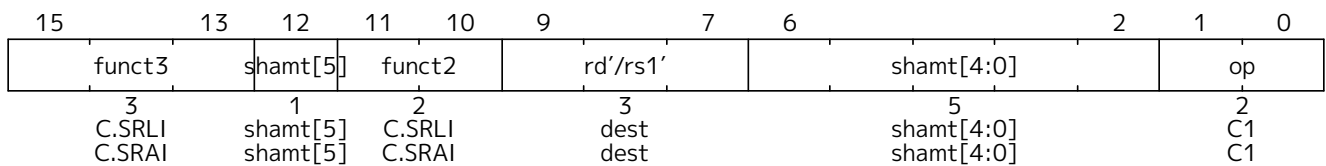
C.ADDI4SPN (add immediate to stack pointer, non-destructive) is a CIW-format instruction that adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, `x2`, and writes the result to `rd'`. This instruction is used to generate pointers to stack-allocated variables, and expands to `addi rd', x2, nzuimm[9:2]`. C.ADDI4SPN is valid only when $nzuimm \neq 0$; 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 `srlui rd', rd', shamt`.

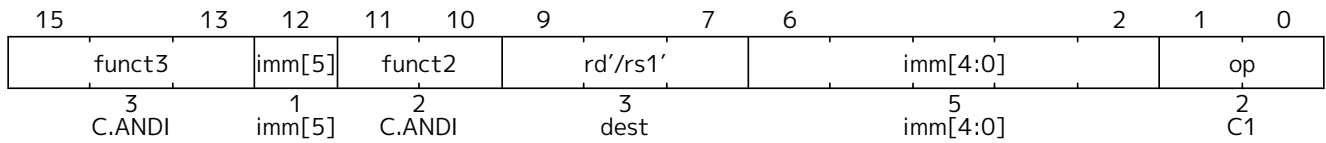
The C.SRLI code points with $shamt = 0$ are HINTs.

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`.

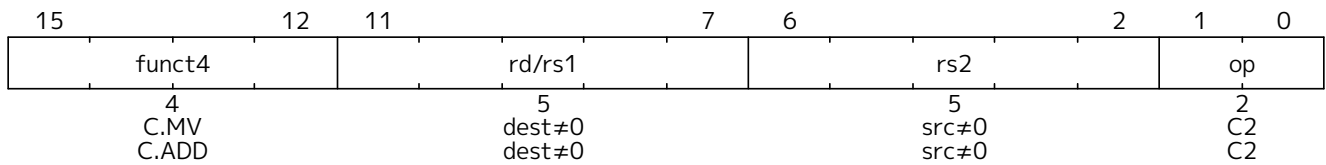


Left shifts are usually more frequent than right shifts, as left shifts are frequently used to scale address values. Right shifts have therefore been granted less encoding space and are placed in an encoding quadrant where all other immediates are sign-extended.



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`.

25.5.3. 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'`.

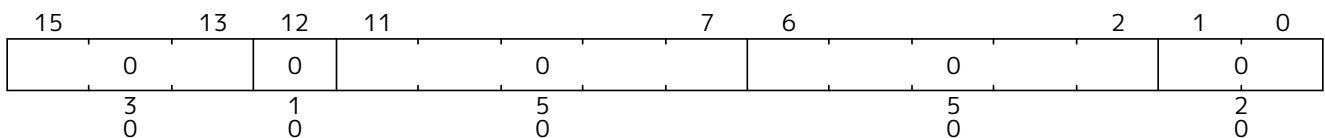
C.ADDW is an RV64C-only instruction that adds the values in registers `rd'` and `rs2'`, then sign-extends the lower 32 bits of the sum before writing the result to register `rd'`. **C.ADDW** expands into `addw rd', rd', rs2'`.

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.

25.5.4. Defined Illegal Instruction

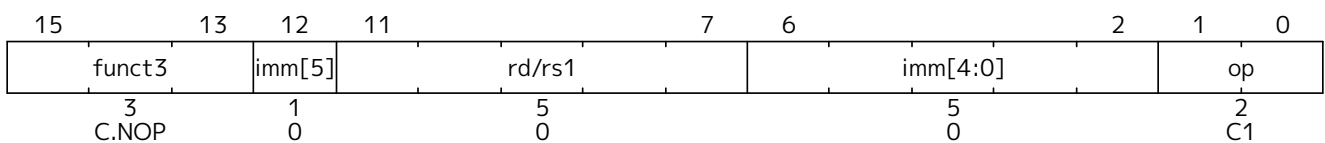


A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.



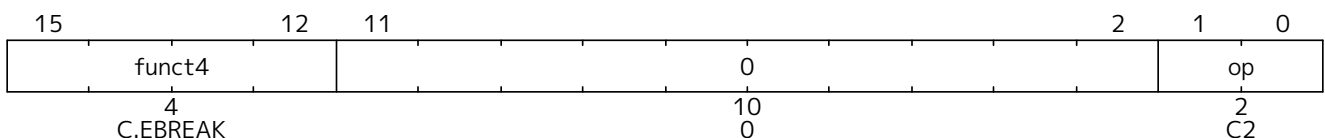
We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zeroed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very long instructions in the RISC-V variable-length encoding scheme) as illegal to capture another common value seen in non-existent memory regions.

25.5.5. 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.

25.5.6. 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.

25.6. 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.

25.7. 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	<i>imm</i> ≠0	63	Designated for future standard use
C.ADDI	<i>rd</i> ≠ <i>x0</i> , <i>imm</i> =0	31	
C.LI	<i>rd</i> = <i>x0</i>	64	
C.LUI	<i>rd</i> = <i>x0</i> , <i>imm</i> ≠0	63	
C.MV	<i>rd</i> = <i>x0</i> , <i>rs2</i> ≠ <i>x0</i>	31	
C.ADD	<i>rd</i> = <i>x0</i> , <i>rs2</i> ≠ <i>x0</i> , <i>rs2</i> ≠ <i>x2</i> - <i>x5</i>	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)	<i>Designated for custom use</i>
C.SRLI	$imm=0$	8	
C.SRAI	$imm=0$	8	

25.8. 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 25.7](#)).

DRAFT

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	14	13	12	11	10	9	8	7	6	5	4	3	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	14	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														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														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	14	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 26. "Zc*" Extension for Code Size Reduction, Version 1.0.0

26.1. 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 <i>c.fsdsp</i> . 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 <i>c.fsdsp</i> . Complex operations intended for embedded CPUs						
cm.jt						yes
cm.jalt						yes

26.2. 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

26.3. 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

26.4. 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

26.5. 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

26.6. 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.

26.7. 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.

26.8. 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	<i>c.lbu rd', uimm(rs1')</i>	Load unsigned byte, 16-bit encoding

RV32	RV64	Mnemonic	Instruction
yes	yes	c.lhu <i>rd'</i> , uimm(<i>rs1'</i>)	Load unsigned halfword, 16-bit encoding
yes	yes	c.lh <i>rd'</i> , uimm(<i>rs1'</i>)	Load signed halfword, 16-bit encoding
yes	yes	c.sb <i>rs2'</i> , uimm(<i>rs1'</i>)	Store byte, 16-bit encoding
yes	yes	c.sh <i>rs2'</i> , uimm(<i>rs1'</i>)	Store halfword, 16-bit encoding
yes	yes	c.zext.b <i>rsd'</i>	Zero extend byte, 16-bit encoding
yes	yes	c.sext.b <i>rsd'</i>	Sign extend byte, 16-bit encoding
yes	yes	c.zext.h <i>rsd'</i>	Zero extend halfword, 16-bit encoding
yes	yes	c.sext.h <i>rsd'</i>	Sign extend halfword, 16-bit encoding
	yes	c.zext.w <i>rsd'</i>	Zero extend word, 16-bit encoding
yes	yes	c.not <i>rsd'</i>	Bitwise not, 16-bit encoding
yes	yes	c.mul <i>rsd'</i> , <i>rs2'</i>	Multiply, 16-bit encoding

DRAFT

26.9. 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

26.10. 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 [Section 26.14.3](#) 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 <i>index</i>	Jump via table
yes	yes	cm.jalt <i>index</i>	Jump and link via table

26.11. 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.

26.12. Zcb instructions

26.12.1. c.lbu

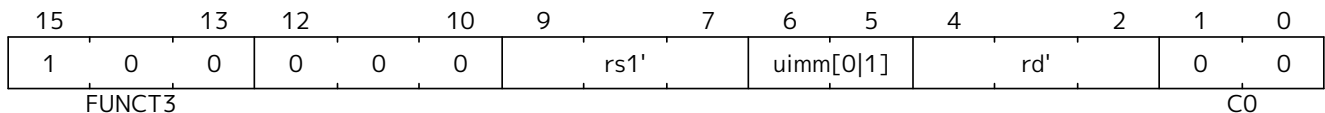
Synopsis

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]);
```


26.12.2. c.lhu

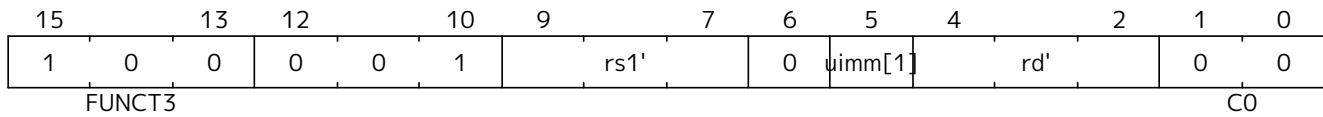
Synopsis

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]);
```

26.12.3. c.lh

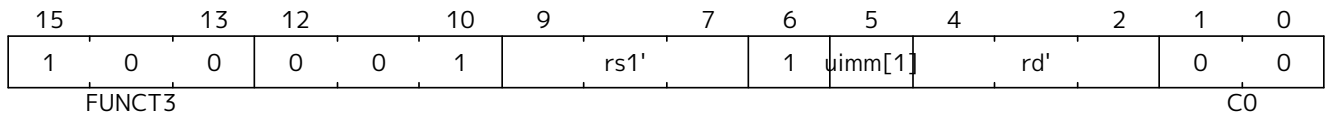
Synopsis

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) = EXTZ(load_mem[X(rs1c)+EXTZ(uimm)] [15..0]);
```

26.12.4. c.sb

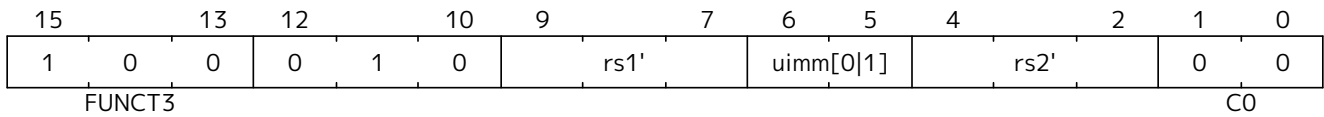
Synopsis

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)
```

26.12.5. c.sh

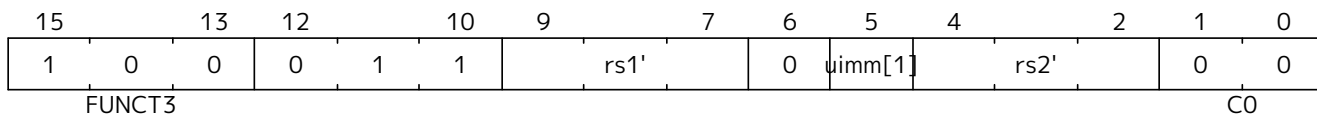
Synopsis

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)
```

26.12.6. c.zext.b

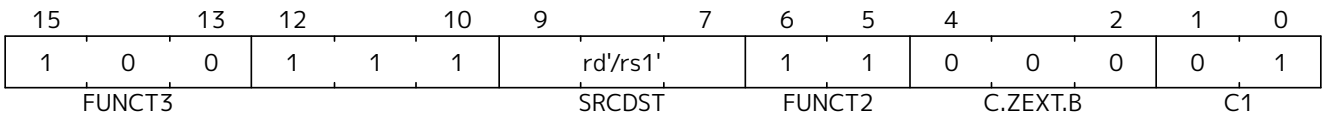
Synopsis

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]);
```

26.12.7. c.sext.b

Synopsis

Sign extend byte, 16-bit encoding

Mnemonic

`c.sext.b rd'/rs1'`

Encoding (RV32, RV64):

15	13	12	10	9	7	6	5	4	2	1	0			
1	0	0	1	1	1	rd'/rs1'		1	1	0	0	1	0	1
FUNCT3			SRCDST			FUNCT2		C.SEXT.B			C1			

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) = EXT(S(X(rsdc)[7..0]);
```

26.12.8. c.zext.h

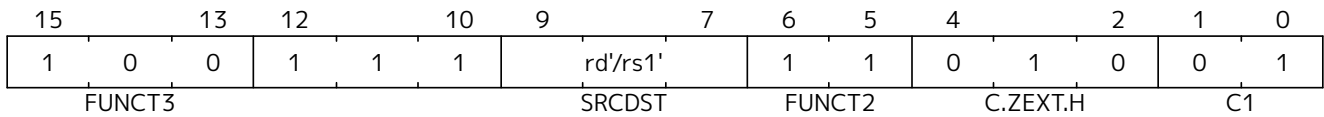
Synopsis

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]);
```


26.12.9. c.sext.h

Synopsis

Sign extend halfword, 16-bit encoding

Mnemonic

`c.sext.h rd'/rs1'`

Encoding (RV32, RV64):

15	13	12	10	9	7	6	5	4	2	1	0			
1	0	0	1	1	1	rd'/rs1'		1	1	0	1	1	0	1
FUNCT3			SRC DST			FUNCT2			C.SEXT.H			C1		

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) = EXT(S(X(rsdc)[15..0]));
```

26.12.10. c.zext.w

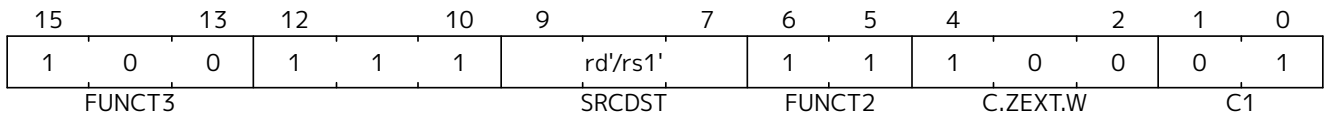
Synopsis

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]);
```

26.12.11. c.not

Synopsis

Bitwise not, 16-bit encoding

Mnemonic

c.not *rd'/rs1'*

Encoding (RV32, RV64):

15	13	12	10	9	7	6	5	4	2	1	0			
1	0	0	1	1	1	rd'/rs1'		1	1	1	0	1	0	1
FUNCT3			SRCDST			FUNCT2			C.NOT			C1		

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;
```

26.12.12. c.mul

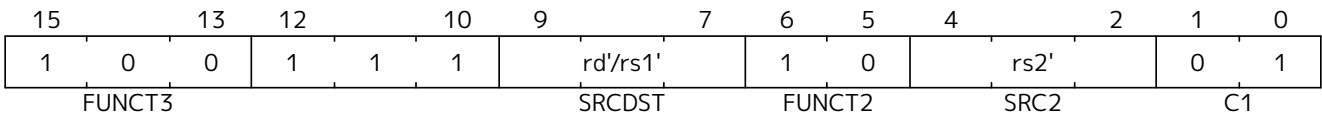
Synopsis

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];
```

26.13. 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.

26.13.1. 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

26.13.2. 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.

26.13.2.1. 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.

26.13.2.2. 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.

26.13.3. PUSH/POP Fault handling

Correct execution requires that *sp* refers to idempotent memory (also see [Section 26.13.5](#)), 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.

26.13.4. Software view of execution

26.13.4.1. 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
```

26.13.4.2. 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
```

26.13.5. 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.

26.13.6. 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.

26.13.6.1. *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;
```

26.13.6.2. *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;
```

26.13.6.3. cm.pop {ra}, 16Encoding: *rlist*=4, *spimm*=0

expands to:

```
lw    ra, 12(sp);  
addi  sp, sp, 16;
```

26.13.6.4. cm.pop [{ra, s0-s3}, 48Encoding: *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;
```

26.13.6.5. cm.pop [{ra, s0-s4}, 64Encoding: *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;
```

26.13.7. cm.push

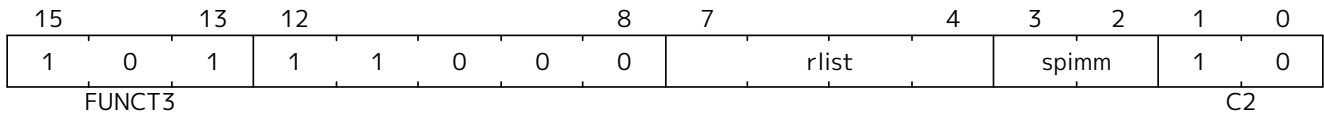
Synopsis

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];  
}
```

DRAFT

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 [Section 26.13](#).

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;
```

DRAFT

26.13.8. cm.pop

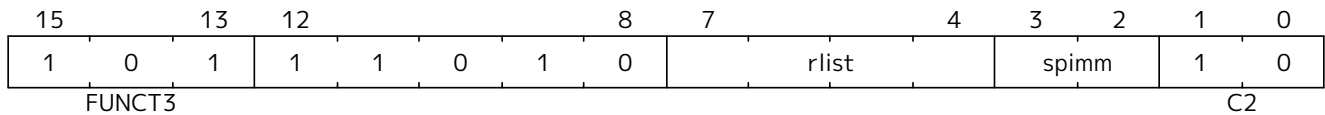
Synopsis

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];  
}
```

DRAFT

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 [Section 26.13](#).

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;
```

DRAFT

26.13.9. cm.popretz

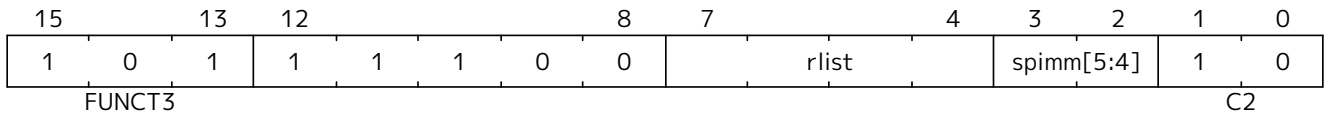
Synopsis

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];  
}
```

DRAFT

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into aO 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 [Section 26.13](#).

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 aO, 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");
```

DRAFT

26.13.10. cm.popret

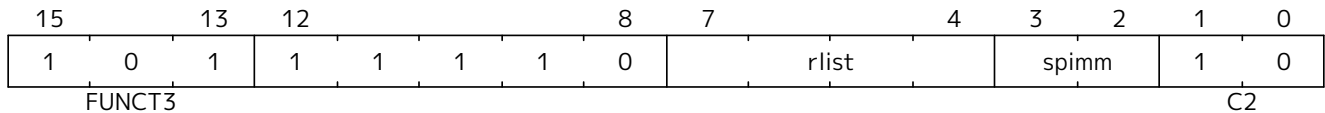
Synopsis

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];  
}
```

DRAFT

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 [Section 26.13](#).

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");
```

DRAFT

26.13.11. cm.mvsa01

Synopsis

Move a0-a1 into two registers of s0-s7

Mnemonic

cm.mvsa01 *r1s', r2s'*

Encoding (RV32, RV64):

15	13	12	10	9	7	6	5	4	2	1	0		
1	0	1	0	1	1	r1s'		0	1	r2s'		1	0
FUNCT3						C2							



For the encoding to be legal $r1s' \neq r2s'$.

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];
```

26.13.12. cm.mva01s

Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

cm.mva01s *r1s'*, *r2s'*

Encoding (RV32, RV64):

15	13	12	10	9	7	6	5	4	2	1	0		
1	0	1	0	1	1	r1s'		1	1	r2s'		1	0
FUNCT3											C2		

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];
```

26.14. 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.

26.14.1. jvt

The base of the table is in the jvt CSR (see [Section 26.14.3](#)), 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.

26.14.2. 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.

26.14.3. jvt CSR

Synopsis

Table jump base vector and control register

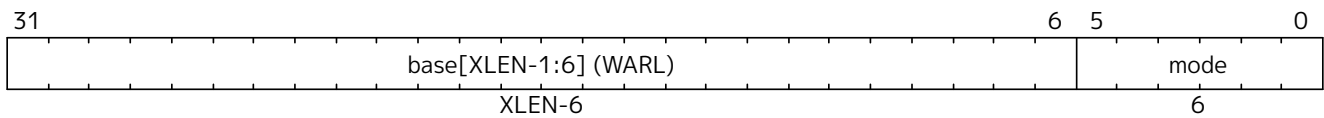
Address:

0x017

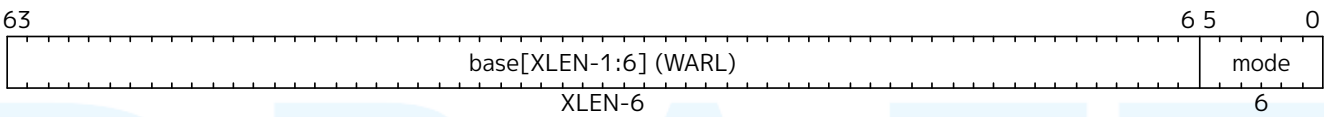
Permissions:

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 26.10](#) 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. <<<

26.14.4. 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 [Section 26.14.5](#).



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 [Section 26.14](#).

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.
```

DRAFT

26.14.5. cm.jalt

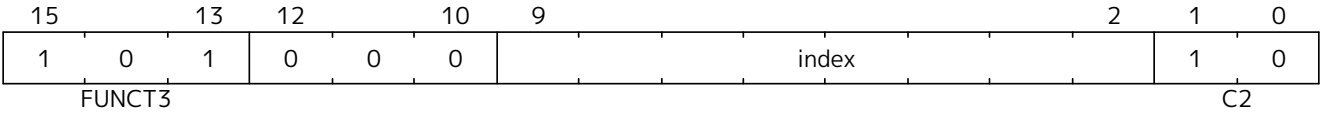
Synopsis

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 [Section 26.14.4](#).



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 [Section 26.14](#).

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.
```

DRAFT

Chapter 27. 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](#)
- Zbkbc: [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	Zbcb	Zbkbc
✓	✓	andn <i>rd, rs1, rs2</i>	AND with inverted operand	✓	✓		
✓	✓	brev8 <i>rd, rs</i>	Reverse bits in bytes		✓		
✓	✓	clmul <i>rd, rs1, rs2</i>	Carry-less multiply (low-part)			✓	✓
✓	✓	clmulh <i>rd, rs1, rs2</i>	Carry-less multiply (high-part)			✓	✓
✓	✓	clmulr <i>rd, rs1, rs2</i>	Carry-less multiply (reversed)			✓	
✓	✓	clz <i>rd, rs</i>	Count leading zero bits	✓			

RV32	RV64	Mnemonic	Instruction	Zbb	Zbk b	Zbc	Zbk c
	✓	clzw <i>rd, rs</i>	Count leading zero bits in word	✓			
✓	✓	cpop <i>rd, rs</i>	Count set bits	✓			
	✓	cpopw <i>rd, rs</i>	Count set bits in word	✓			
✓	✓	ctz <i>rd, rs</i>	Count trailing zero bits	✓			
	✓	ctzw <i>rd, rs</i>	Count trailing zero bits in word	✓			
✓	✓	max <i>rd, rs1, rs2</i>	Maximum	✓			
✓	✓	maxu <i>rd, rs1, rs2</i>	Unsigned maximum	✓			
✓	✓	min <i>rd, rs1, rs2</i>	Minimum	✓			
✓	✓	minu <i>rd, rs1, rs2</i>	Unsigned minimum	✓			
✓	✓	orc.b <i>rd, rs</i>	Bitwise OR-Combine, byte granule	✓			
✓	✓	orn <i>rd, rs1, rs2</i>	OR with inverted operand	✓	✓		
✓	✓	pack <i>rd, rs1, rs2</i>	Pack low halves of registers		✓		
✓	✓	packh <i>rd, rs1, rs2</i>	Pack low bytes of registers		✓		
	✓	packw <i>rd, rs1, rs2</i>	Pack low 16-bits of registers (RV64)		✓		
✓	✓	rev8 <i>rd, rs</i>	Byte-reverse register	✓	✓		
✓	✓	rol <i>rd, rs1, rs2</i>	Rotate left (Register)	✓	✓		
	✓	rolw <i>rd, rs1, rs2</i>	Rotate Left Word (Register)	✓	✓		
✓	✓	ror <i>rd, rs1, rs2</i>	Rotate right (Register)	✓	✓		
✓	✓	rori <i>rd, rs1, shamt</i>	Rotate right (Immediate)	✓	✓		
	✓	roriw <i>rd, rs1, shamt</i>	Rotate right Word (Immediate)	✓	✓		
	✓	rorw <i>rd, rs1, rs2</i>	Rotate right Word (Register)	✓	✓		
✓	✓	sext.b <i>rd, rs</i>	Sign-extend byte	✓			
✓	✓	sext.h <i>rd, rs</i>	Sign-extend halfword	✓			
✓		unzip <i>rd, rs</i>	Bit deinterleave		✓		
✓	✓	xnor <i>rd, rs1, rs2</i>	Exclusive NOR	✓	✓		
✓	✓	zext.h <i>rd, rs</i>	Zero-extend halfword	✓			
✓		zip <i>rd, rs</i>	Bit interleave		✓		

RV32	RV64	Mnemonic	Instruction	Zba	Zbs
	✓	add.uw <i>rd, rs1, rs2</i>	Add unsigned word	✓	
✓	✓	bclr <i>rd, rs1, rs2</i>	Single-Bit Clear (Register)		✓
✓	✓	bclri <i>rd, rs1, imm</i>	Single-Bit Clear (Immediate)		✓
✓	✓	bext <i>rd, rs1, rs2</i>	Single-Bit Extract (Register)		✓
✓	✓	bexti <i>rd, rs1, imm</i>	Single-Bit Extract (Immediate)		✓
✓	✓	binv <i>rd, rs1, rs2</i>	Single-Bit Invert (Register)		✓
✓	✓	binvi <i>rd, rs1, imm</i>	Single-Bit Invert (Immediate)		✓
✓	✓	bset <i>rd, rs1, rs2</i>	Single-Bit Set (Register)		✓
✓	✓	bseti <i>rd, rs1, imm</i>	Single-Bit Set (Immediate)		✓
✓	✓	sh1add <i>rd, rs1, rs2</i>	Shift left by 1 and add	✓	

RV32	RV64	Mnemonic	Instruction	Zba	Zbs
	✓	sh1add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 1 and add	✓	
✓	✓	sh2add <i>rd, rs1, rs2</i>	Shift left by 2 and add	✓	
	✓	sh2add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 2 and add	✓	
✓	✓	sh3add <i>rd, rs1, rs2</i>	Shift left by 3 and add	✓	
	✓	sh3add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 3 and add	✓	
	✓	slli.uw <i>rd, rs1, imm</i>	Shift-left unsigned word (Immediate)	✓	

27.1. "B" Extension for Bit Manipulation, Version 1.0.0

The B standard extension comprises instructions provided by the Zba, Zbb, and Zbs extensions.

27.2. 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 <i>rd, rs1, rs2</i>	Add unsigned word
✓	✓	sh1add <i>rd, rs1, rs2</i>	Shift left by 1 and add
	✓	sh1add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 1 and add
✓	✓	sh2add <i>rd, rs1, rs2</i>	Shift left by 2 and add
	✓	sh2add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 2 and add
✓	✓	sh3add <i>rd, rs1, rs2</i>	Shift left by 3 and add
	✓	sh3add.uw <i>rd, rs1, rs2</i>	Shift unsigned word left by 3 and add
	✓	slli.uw <i>rd, rs1, imm</i>	Shift-left unsigned word (Immediate)

27.3. Zbb: Extension for Basic bit-manipulation, Version 1.0.0

27.3.1. Logical with negate

RV32	RV64	Mnemonic	Instruction
✓	✓	andn <i>rd, rs1, rs2</i>	AND with inverted operand
✓	✓	orn <i>rd, rs1, rs2</i>	OR with inverted operand

RV32	RV64	Mnemonic	Instruction
✓	✓	xnor <i>rd, rs1, rs2</i>	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.

27.3.2. Count leading/trailing zero bits

RV32	RV64	Mnemonic	Instruction
✓	✓	clz <i>rd, rs</i>	Count leading zero bits
	✓	clzw <i>rd, rs</i>	Count leading zero bits in word
✓	✓	ctz <i>rd, rs</i>	Count trailing zero bits
	✓	ctzw <i>rd, rs</i>	Count trailing zero bits in word

27.3.3. 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 <i>rd, rs</i>	Count set bits
	✓	cpopw <i>rd, rs</i>	Count set bits in word

27.3.4. 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 <i>rd, rs1, rs2</i>	Maximum
✓	✓	maxu <i>rd, rs1, rs2</i>	Unsigned maximum
✓	✓	min <i>rd, rs1, rs2</i>	Minimum
✓	✓	minu <i>rd, rs1, rs2</i>	Unsigned minimum

27.3.5. 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>) + srli` (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 <i>rd, rs</i>	Sign-extend byte

RV32	RV64	Mnemonic	Instruction
✓	✓	sext.h rd, rs	Sign-extend halfword
✓	✓	zext.h rd, rs	Zero-extend halfword

27.3.6. 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*)

27.3.7. 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

27.3.8. Byte-reverse

rev8 reverses the byte-ordering of *rs*.

RV32	RV64	Mnemonic	Instruction
✓	✓	rev8 rd, rs	Byte-reverse register

27.4. 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 <i>rd, rs1, rs2</i>	Carry-less multiply (low-part)
✓	✓	clmulh <i>rd, rs1, rs2</i>	Carry-less multiply (high-part)
✓	✓	clmulr <i>rd, rs1, rs2</i>	Carry-less multiply (reversed)

27.5. 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 <i>rd, rs1, rs2</i>	Single-Bit Clear (Register)
✓	✓	bclri <i>rd, rs1, imm</i>	Single-Bit Clear (Immediate)
✓	✓	bext <i>rd, rs1, rs2</i>	Single-Bit Extract (Register)
✓	✓	bexti <i>rd, rs1, imm</i>	Single-Bit Extract (Immediate)
✓	✓	binv <i>rd, rs1, rs2</i>	Single-Bit Invert (Register)
✓	✓	binvi <i>rd, rs1, imm</i>	Single-Bit Invert (Immediate)
✓	✓	bset <i>rd, rs1, rs2</i>	Single-Bit Set (Register)
✓	✓	bseti <i>rd, rs1, imm</i>	Single-Bit Set (Immediate)

27.6. 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	Section 27.9.31
	✓	rolw	Section 27.9.32
✓	✓	ror	Section 27.9.33
✓	✓	rori	Section 27.9.34
	✓	roriw	Section 27.9.35
	✓	rorw	Section 27.9.36
✓	✓	andn	Section 27.9.2
✓	✓	orn	Section 27.9.25
✓	✓	xnor	Section 27.9.47
✓	✓	pack	Section 27.9.26
✓	✓	packh	Section 27.9.27
	✓	packw	Section 27.9.28
✓	✓	brev8	Section 27.9.30
✓	✓	rev8	Section 27.9.29
✓		zip	Section 27.9.51
✓		unzip	Section 27.9.46

27.7. Zbk: 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 extension is a subset of the Zbc [Section 27.4](#) extension, and only provides `clmul` and `clmulh`. These are the crucial instructions needed to efficiently implement the GHASH operation, a critical operation in some cryptographic workloads such as the AES-GCM authenticated encryption scheme. See Zbc [Section 27.4](#) for further instruction details for these two instructions.

27.8. 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
✓	✓	<code>xperm4 rd, rs1, rs2</code>	Crossbar permutation (nibbles)
✓	✓	<code>xperm8 rd, rs1, rs2</code>	Crossbar permutation (bytes)

27.9. Instructions (in alphabetical order)

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

27.9.1. add.uw

Synopsis

Add unsigned word

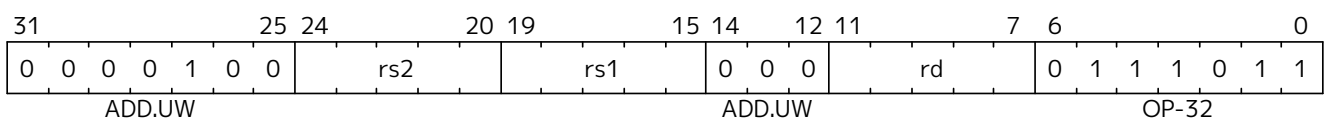
Mnemonic

add.uw *rd*, *rs1*, *rs2*

Pseudoinstructions

zext.w rd, rs1 $\rightarrow$ 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 (Section 27.2)	0.93	Ratified

27.9.2. andn

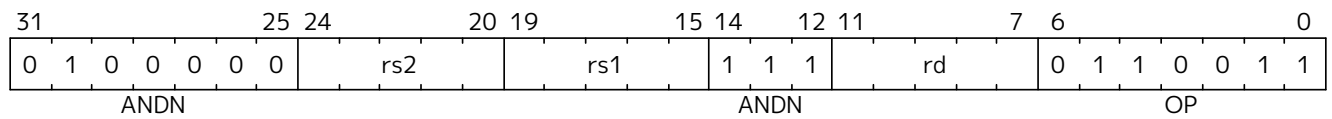
Synopsis

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

27.9.3. bclr

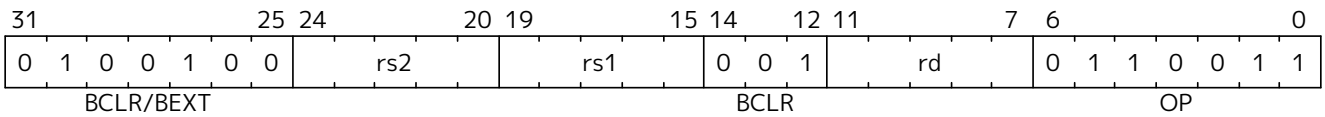
Synopsis

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 $\log_2(\text{XLEN})$ 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

27.9.4. bclri

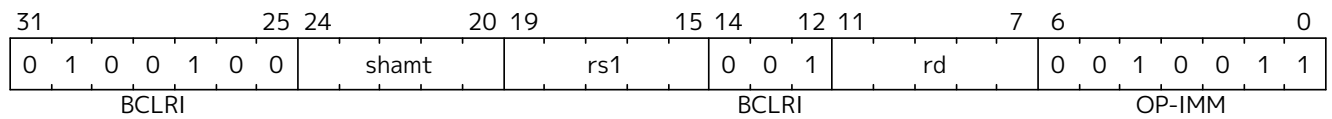
Synopsis

Single-Bit Clear (Immediate)

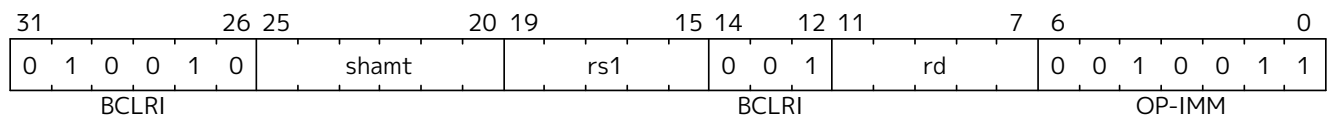
Mnemonic

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

27.9.5. bext

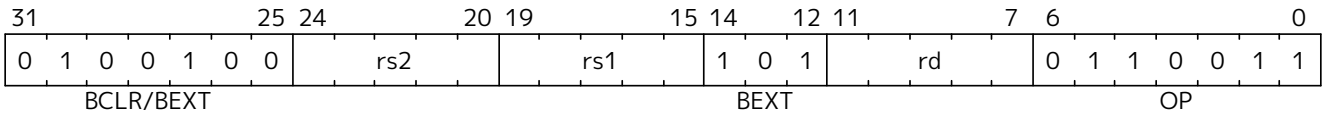
Synopsis

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

27.9.6. bexti

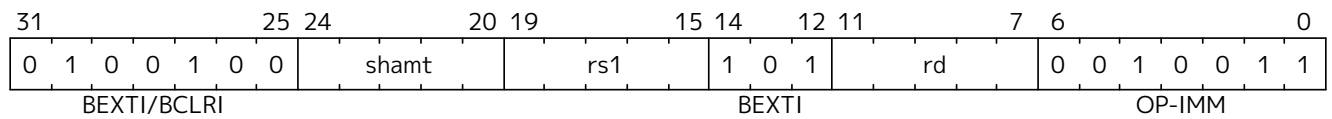
Synopsis

Single-Bit Extract (Immediate)

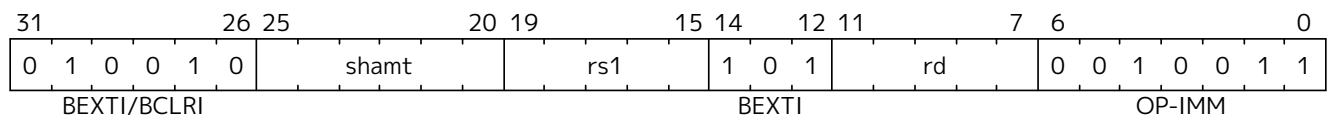
Mnemonic

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

27.9.7. binv

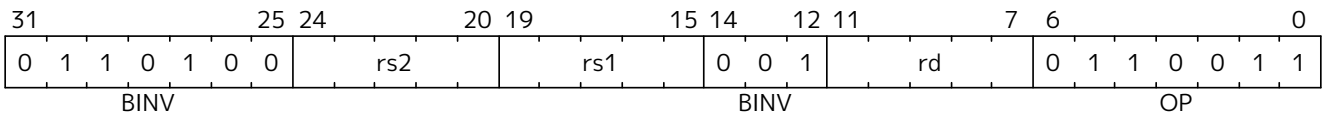
Synopsis

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

27.9.8. binvi

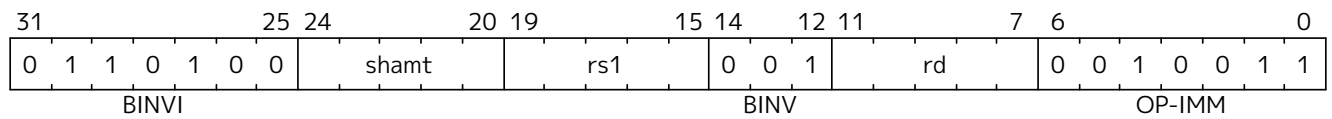
Synopsis

Single-Bit Invert (Immediate)

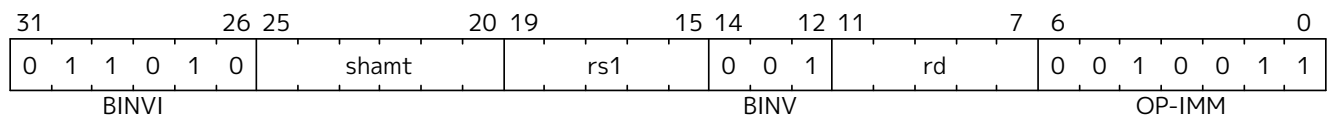
Mnemonic

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 $\log_2(\text{XLEN})$ bits of *shamt*. For RV32, the encodings corresponding to $\text{shamt}[5]=1$ 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

27.9.9. bset

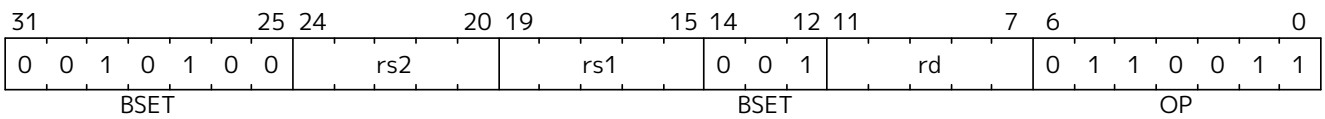
Synopsis

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 $\log_2(\text{XLEN})$ 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

27.9.10. bseti

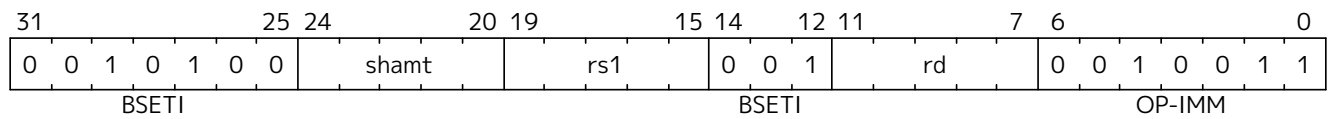
Synopsis

Single-Bit Set (Immediate)

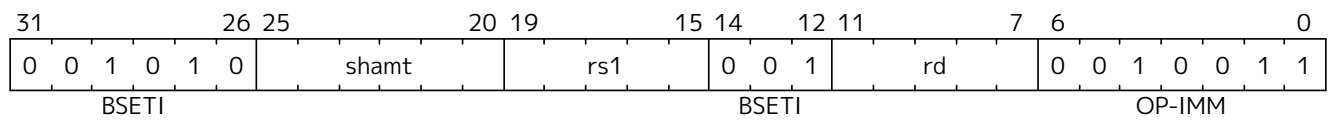
Mnemonic

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 $\log_2(\text{XLEN})$ 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

27.9.11. clmul

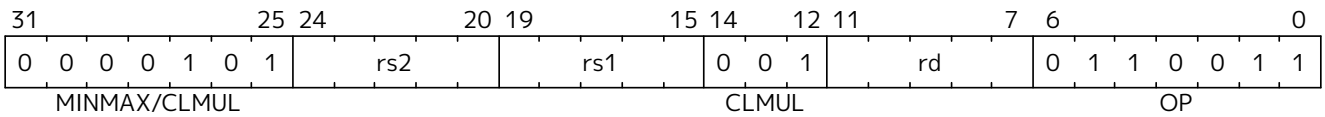
Synopsis

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

27.9.12. clmulh

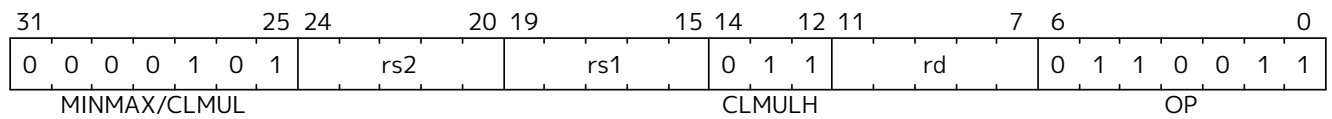
Synopsis

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

27.9.13. clmulr

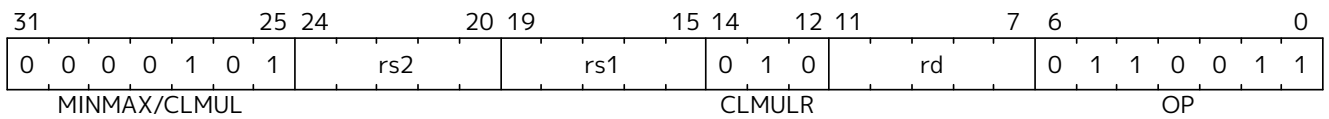
Synopsis

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

27.9.14. clz

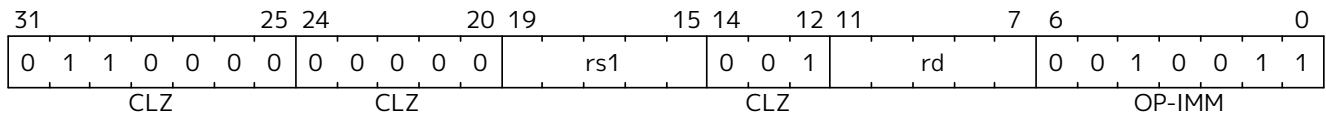
Synopsis

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

27.9.15. clzw

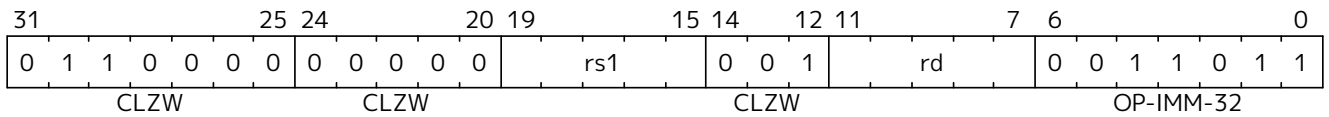
Synopsis

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

27.9.16. cpop

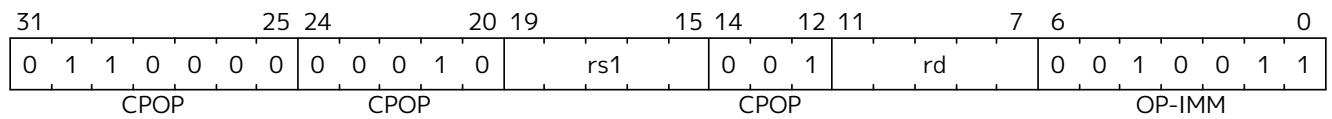
Synopsis

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

27.9.17. cpopw

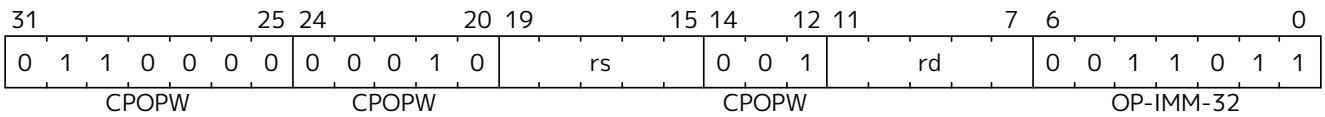
Synopsis

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

27.9.18. ctz

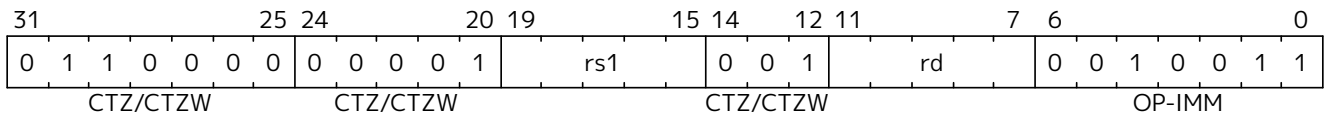
Synopsis

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

27.9.19. ctzw

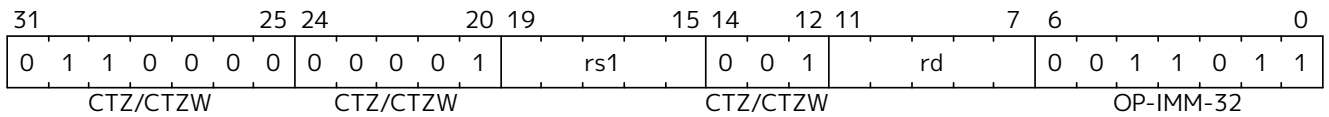
Synopsis

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

27.9.20. max

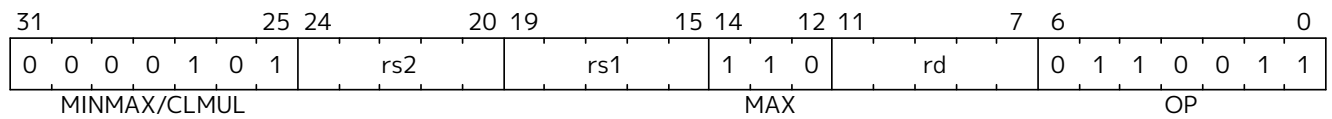
Synopsis

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

27.9.21. maxu

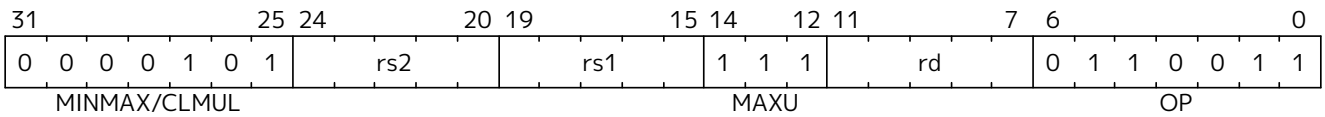
Synopsis

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

27.9.22. min

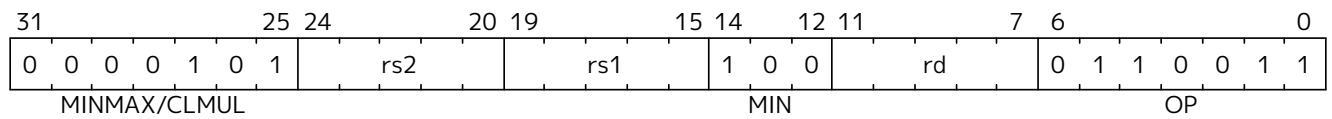
Synopsis

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

27.9.23. minu

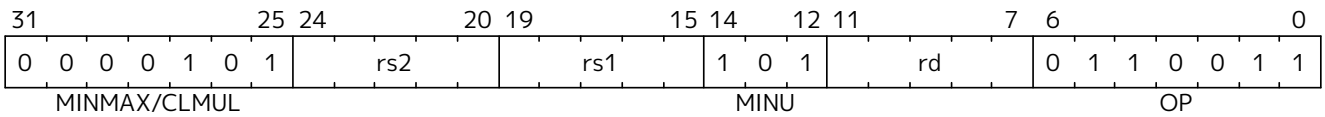
Synopsis

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

27.9.24. orc.b

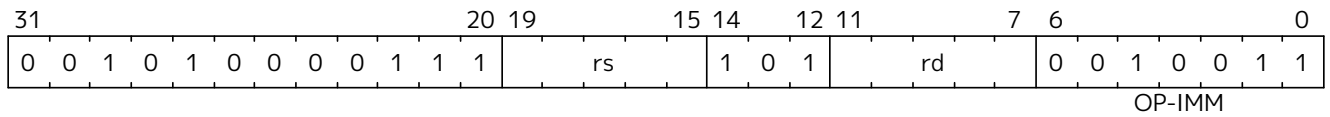
Synopsis

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

27.9.25. orn

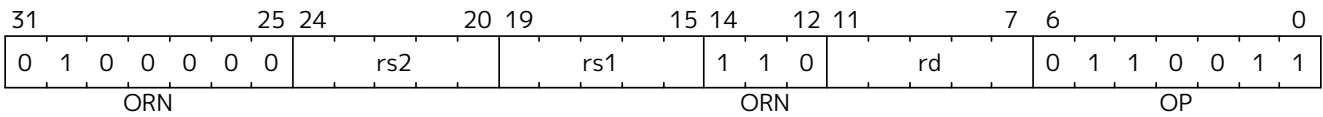
Synopsis

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

27.9.26. pack

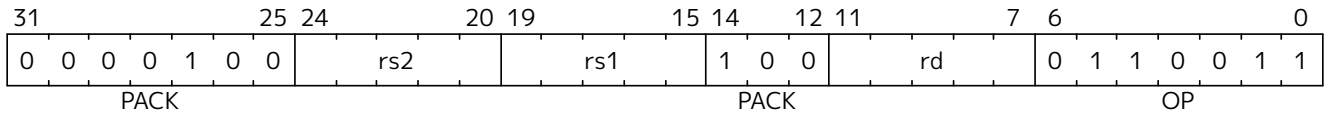
Synopsis

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
Zbbk (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).

27.9.27. packh

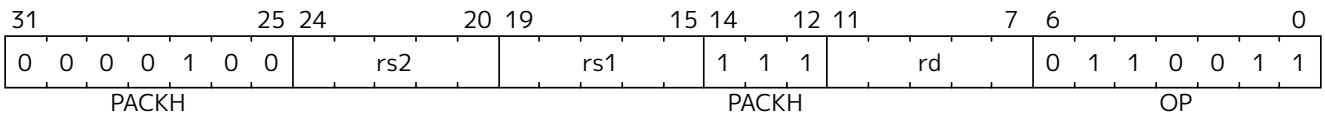
Synopsis

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

27.9.29. rev8

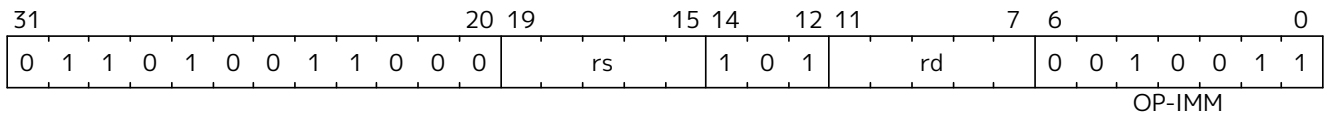
Synopsis

Byte-reverse register

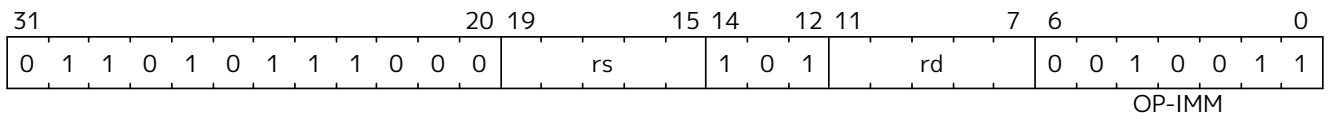
Mnemonic

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
Zbbk (Bit-manipulation for Cryptography)	v1.0	Ratified

27.9.30. brev8

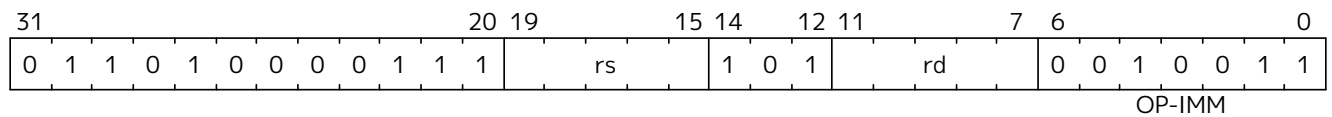
Synopsis

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

27.9.31. rol

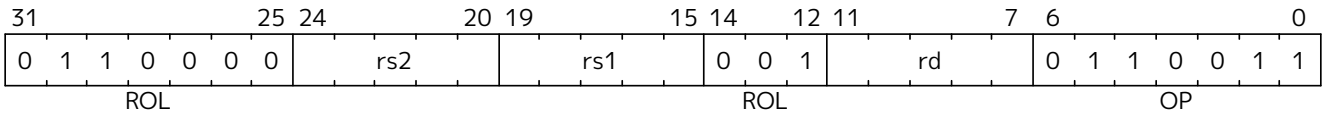
Synopsis

Rotate Left (Register)

Mnemonic

rol *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate left of *rs1* by the amount in least-significant $\log_2(\text{XLEN})$ 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

27.9.32. rolw

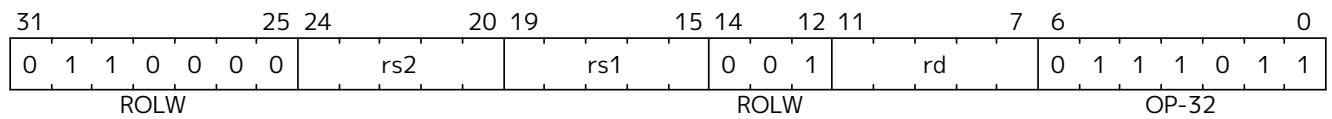
Synopsis

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

27.9.33. ror

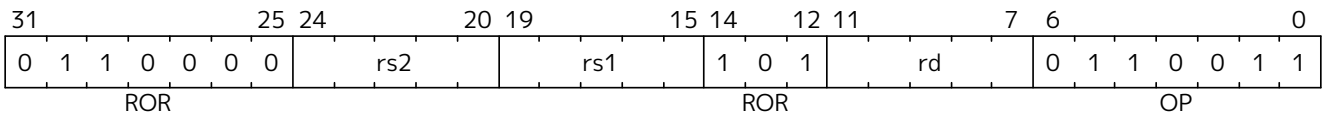
Synopsis

Rotate Right

Mnemonic

ror *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right of *rs1* by the amount in least-significant $\log_2(\text{XLEN})$ 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

27.9.34. rori

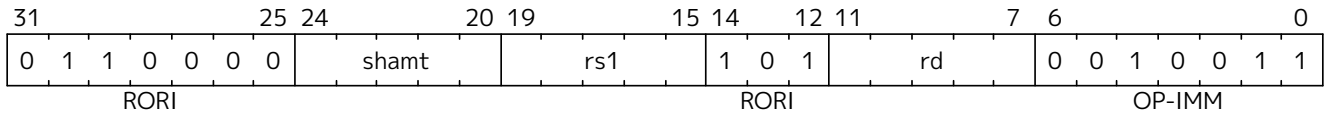
Synopsis

Rotate Right (Immediate)

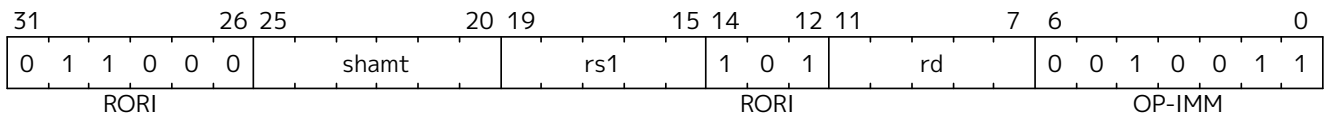
Mnemonic

`rori rd, rs1, shamt`

Encoding (RV32)



Encoding (RV64)



Description

This instruction performs a rotate right of *rs1* by the amount in the least-significant $\log_2(\text{XLEN})$ 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

27.9.35. roriw

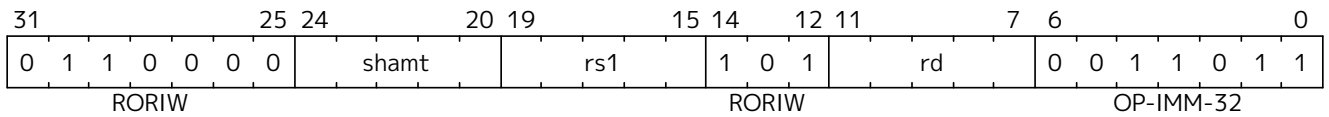
Synopsis

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 $\log_2(\text{XLEN})$ 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

27.9.37. sext.b

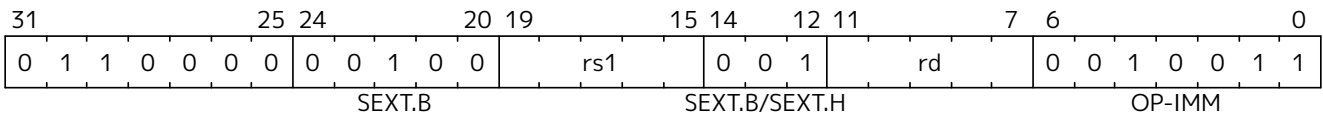
Synopsis

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) = EXT(S(X(rs)[7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified

27.9.38. sext.h

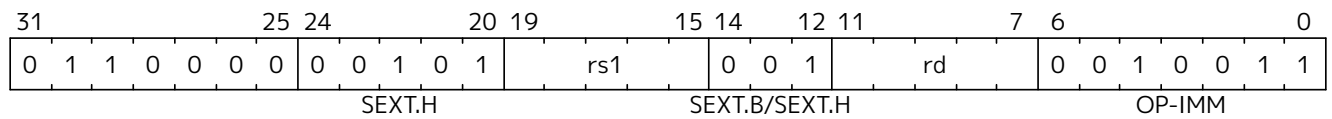
Synopsis

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

27.9.39. sh1add

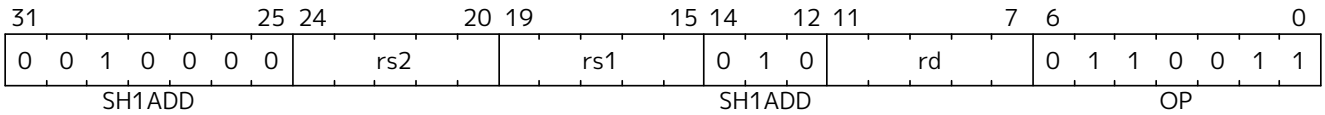
Synopsis

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

27.9.40. sh1add.uw

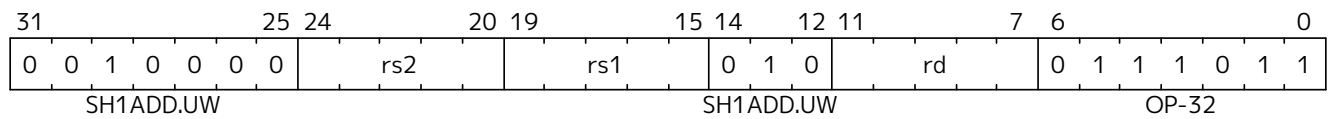
Synopsis

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

27.9.41. sh2add

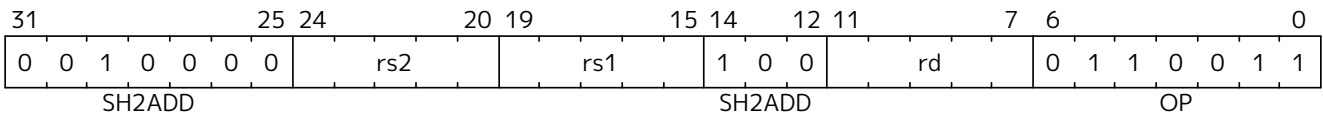
Synopsis

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

27.9.42. sh2add.uw

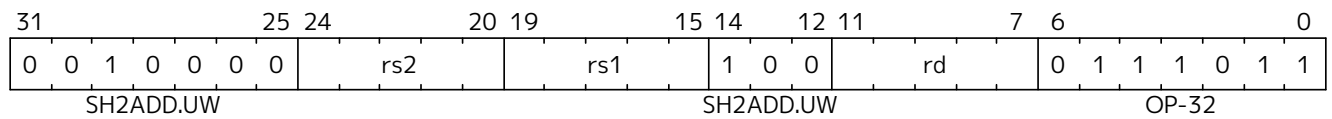
Synopsis

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

27.9.43. sh3add

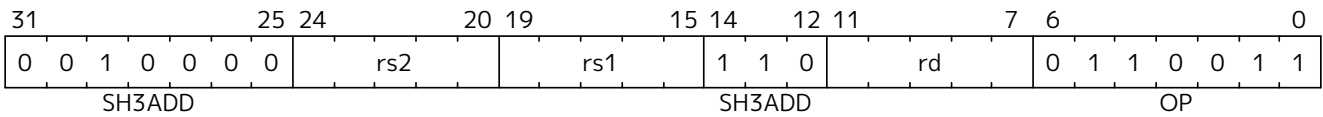
Synopsis

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

27.9.44. sh3add.uw

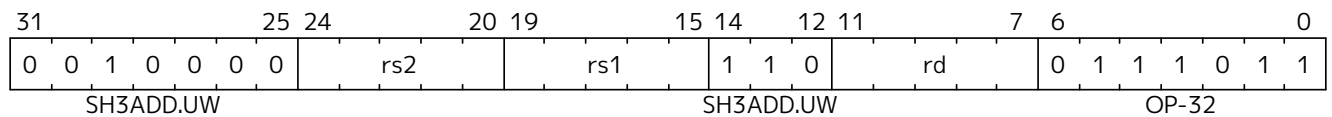
Synopsis

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

27.9.45. slli.uw

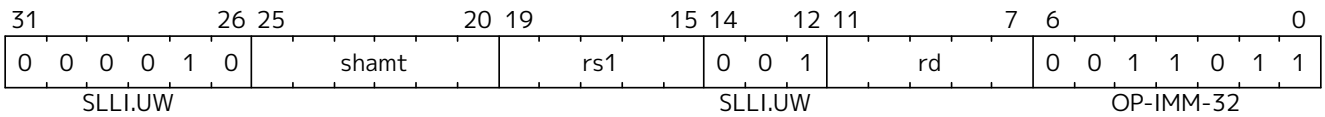
Synopsis

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.

27.9.46. unzip

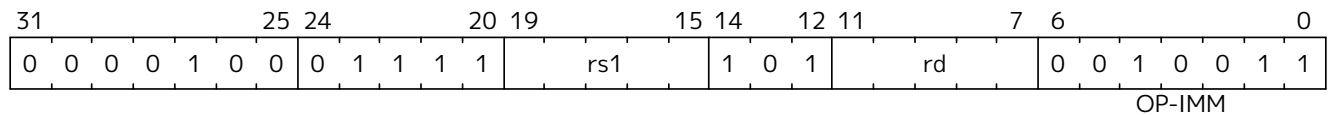
Synopsis

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

27.9.47. xnor

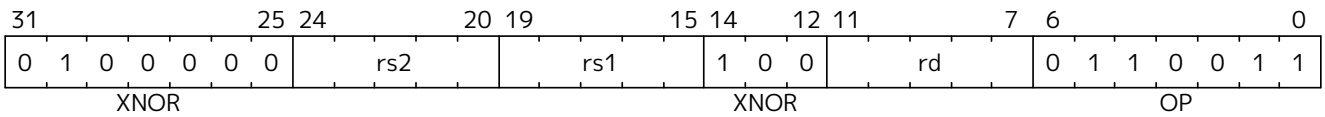
Synopsis

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) = $\sim(X(rs1) \wedge X(rs2))$;

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	0.93	Ratified
Zbkb (Bit-manipulation for Cryptography)	v1.0	Ratified

27.9.49. xperm4

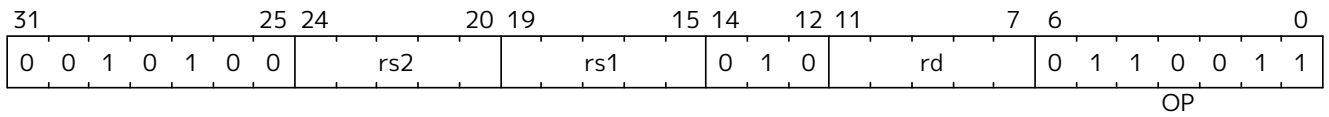
Synopsis

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

27.9.50. zext.h

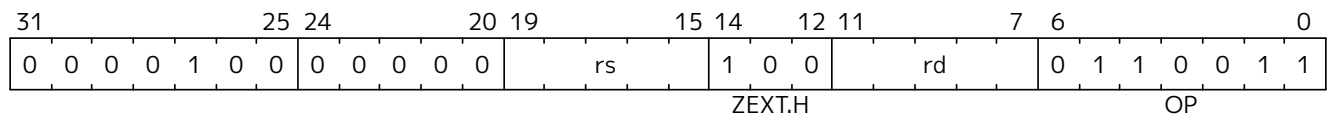
Synopsis

Zero-extend halfword

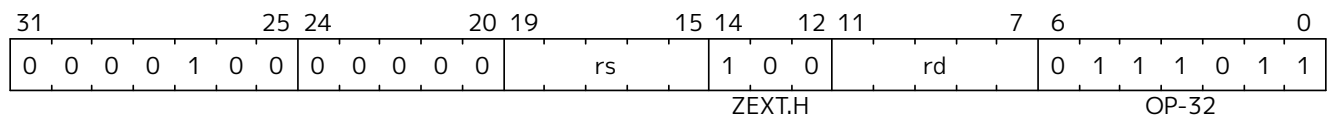
Mnemonic

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

27.9.51. zip

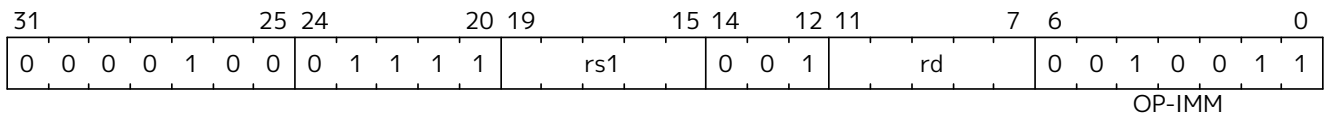
Synopsis

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 28. "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.

28.1. Introduction

Section 28.18 lists the standard vector extensions and which instructions and element widths are supported by each extension.

28.2. 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, $ELEN \geq 8$, which must be a power of 2.
2. The number of bits in a single vector register, $VLEN \geq ELEN$, which must be a power of 2, and must be no greater than 2^{16} .

Standard vector extensions (Section 28.18) 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.

28.3. 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**.

28.3.1. 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.

28.3.2. 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**.

28.3.3. 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 `misal.V` field, the `vsstatus.VS` field may exist even if `misal.V` is clear.

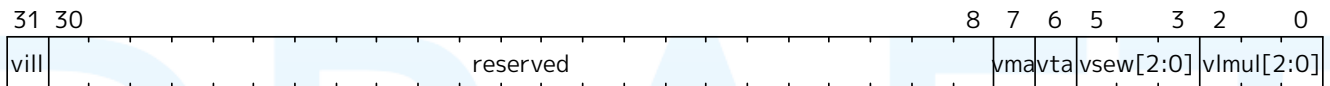
28.3.4. Vector Type (vtype) Register

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	<code>vill</code>	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	<code>vma</code>	Vector mask agnostic
6	<code>vta</code>	Vector tail agnostic
5:3	<code>vsew[2:0]</code>	Selected element width (SEW) setting
2:0	<code>vlmul[2:0]</code>	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.

28.3.4.1. 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.

28.3.4.2. 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 $LMUL \geq SEW_{MIN}/ELEN$, 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., $LMUL = 2^{vlmul[2:0]}$).

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 <i>n</i>
1	0	0	-	-	-	reserved
1	0	1	1/8	32	$VLEN/SEW/8$	v <i>n</i> (single register in group)
1	1	0	1/4	32	$VLEN/SEW/4$	v <i>n</i> (single register in group)

vmlul[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	$2 \times \text{VLEN/SEW}$	$v\ n, v\ n+1$
0	1	0	4	8	$4 \times \text{VLEN/SEW}$	$v\ n, \dots, v\ n+3$
0	1	1	8	4	$8 \times \text{VLEN/SEW}$	$v\ n, \dots, v\ n+7$

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.

28.3.4.3. 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 28.5.4](#).

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.

28.3.4.4. 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.

28.3.5. 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 28.5.4](#).



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 $VLMAX=32$).

28.3.6. 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.

28.3.7. 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 28.5.4](#).

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.*

28.3.8. Vector Fixed-Point Rounding Mode (**vfrm**) Register

The vector fixed-point rounding-mode register holds a two-bit read-write rounding-mode field in the least-significant bits (**vfrm**[1:0]). The upper bits, **vfrm**[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 $(v \gg d) + r$, where **r** depends on the rounding mode as specified in the following table.

Table 46. **vfrm** encoding

vfrm [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.

28.3.9. 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**.

28.3.10. 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

28.3.11. 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.

28.4. 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.

28.4.1. 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

28.4.2. 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	

28.4.3. 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									0
v2*n+1							7									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																0
v4*n+1																4
v4*n+2																8
v4*n+3																C

28.4.4. 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 be used to simply increase vector length to reduce instruction fetch and dispatch overheads in cases where fewer vector register groups are needed.

28.4.5. Mask Register Layout

A vector mask occupies only one vector register regardless of SEW and LMUL.

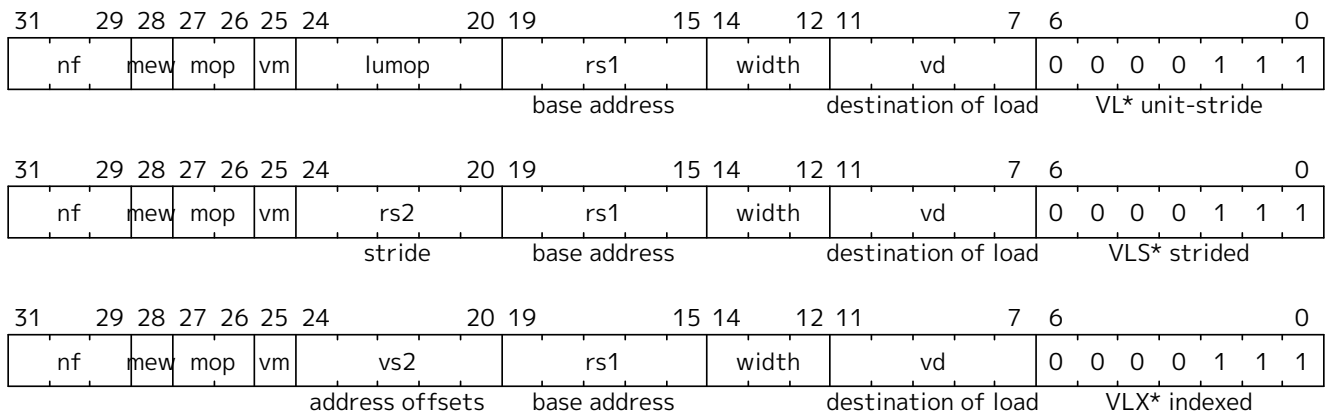
Each element is allocated a single mask bit in a mask vector register. The mask bit for element i is located in bit i of the mask register, independent of SEW or LMUL.

28.5. Vector Instruction Formats

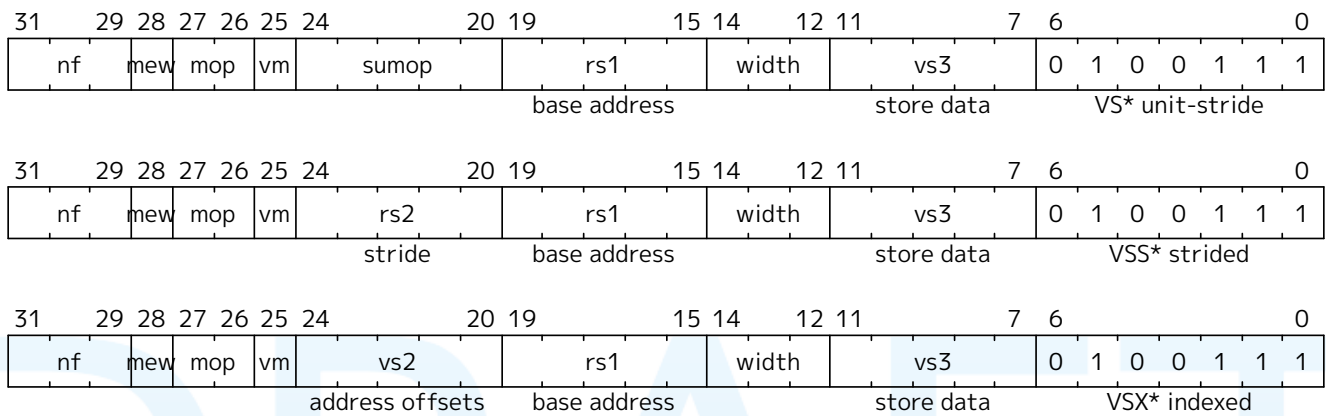
The instructions in the vector extension fit under two existing major opcodes (LOAD-FP and STORE-FP) and one new major opcode (OP-V).

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 28.5.3.1](#)).

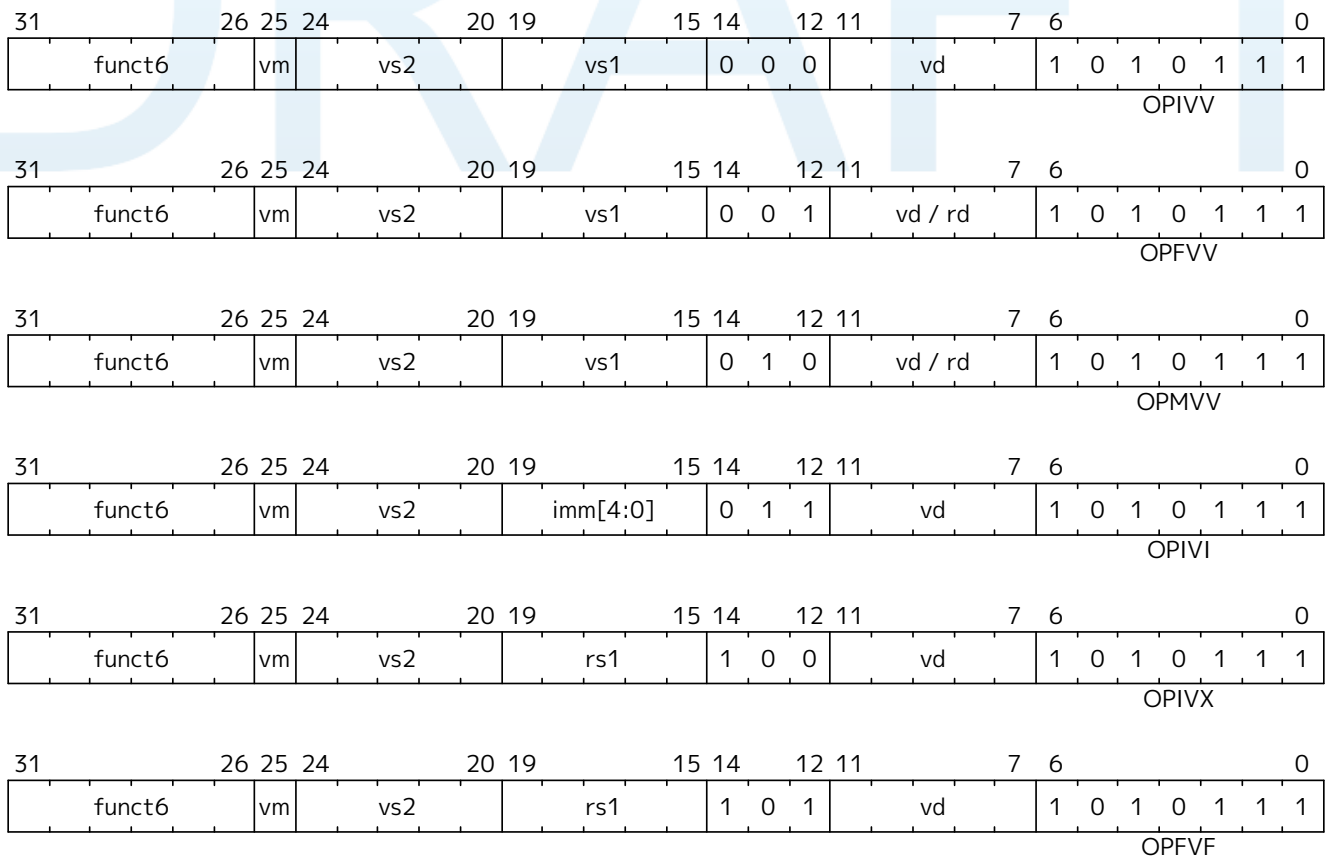
Format for Vector Load Instructions under LOAD-FP major opcode

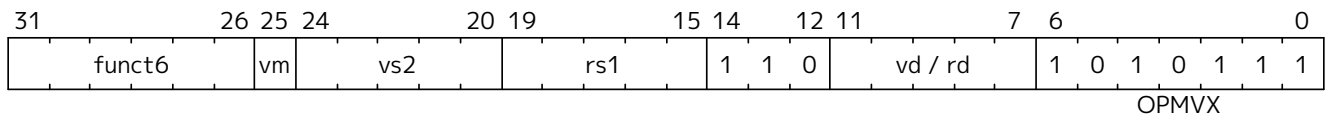


Format for Vector Store Instructions under STORE-FP major opcode

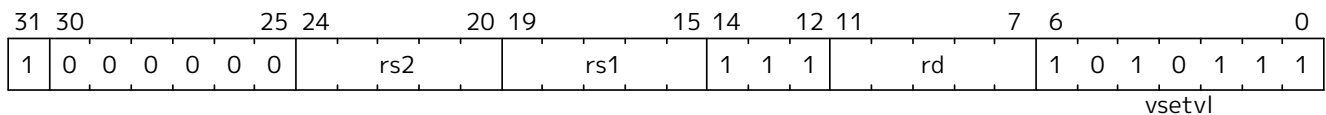
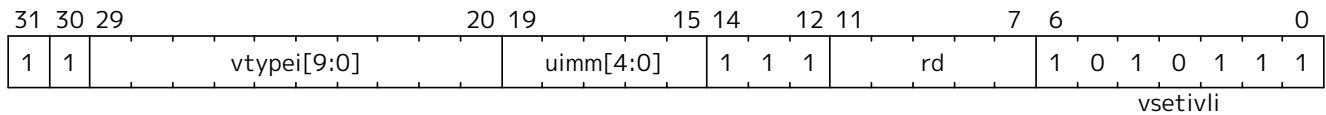
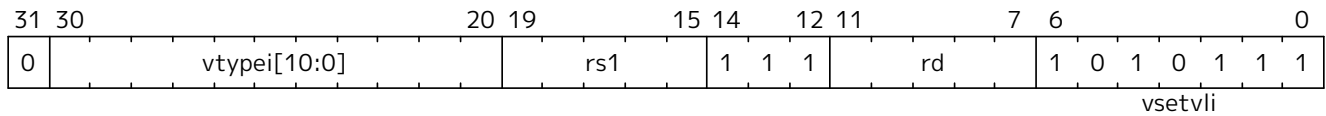


Formats for Vector Arithmetic Instructions under OP-V major opcode





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.

28.5.1. 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.*

28.5.2. 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 EEW s, 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., $EMUL \leq 8$), 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 $EMUL=1$.

28.5.3. 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 28.3.4.3).

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 28.3.4.3](#), mask destination tail elements are always treated as tail-agnostic, regardless of the setting of **vta**.

28.5.3.1. 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]**.*

28.5.4. 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.*

28.6. Configuration-Setting Instructions (**vsetvli**/**vsetivli**/**vsetvl**)

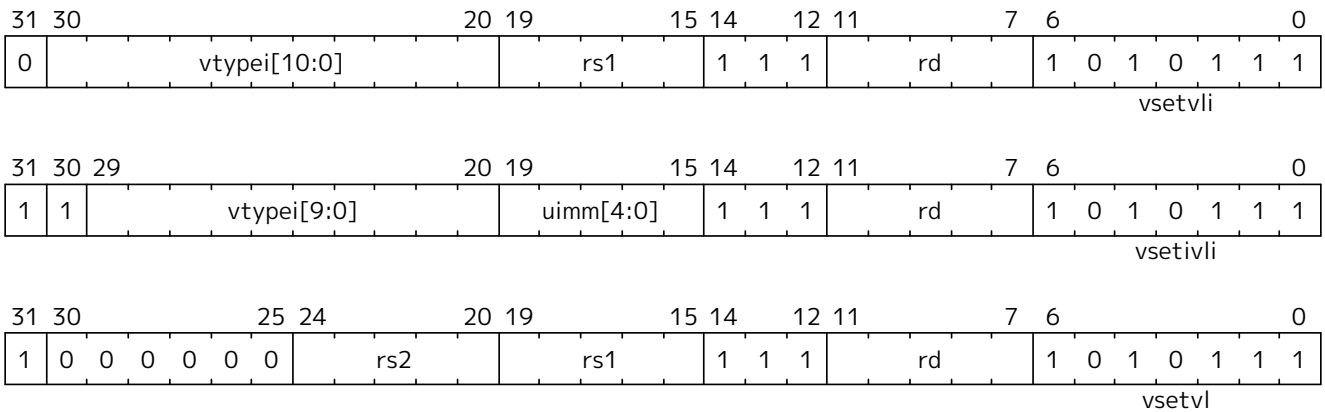
One of the common approaches to handling a large number of elements is "strip mining" where each iteration of a loop handles some number of elements, and the iterations continue until all elements have been processed. The RISC-V vector specification provides direct, portable support for this approach. 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 28.6.4](#), 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



28.6.1. 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	<code>vill</code>	Illegal value if set
XLEN-2:8	0	Reserved if non-zero
7	<code>vma</code>	Vector mask agnostic
6	<code>vta</code>	Vector tail agnostic
5:3	<code>vsew[2:0]</code>	Selected element width (SEW) setting
2:0	<code>vlmul[2:0]</code>	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.

28.6.1.1. 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.

28.6.2. 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.*

28.6.3. 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. $\text{ceil}(\text{AVL} / 2) \leq \text{vl} \leq \text{VLMAX}$ if $\text{AVL} < (2 * \text{VLMAX})$
3. **vl** = VLMAX if $\text{AVL} \geq (2 * \text{VLMAX})$
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 $\text{AVL} \leq \text{VLMAX}$, yet flexible enough to enable implementations to improve vector lane utilization for $\text{AVL} > \text{VLMAX}$.*

*For example, this permits an implementation to set **vl** = $\text{ceil}(\text{AVL} / 2)$ for $\text{VLMAX} < \text{AVL} <$*

*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 $VLMAX < AVL < 2*VLMAX$*

28.6.4. 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?
```

28.7. Vector Loads and Stores

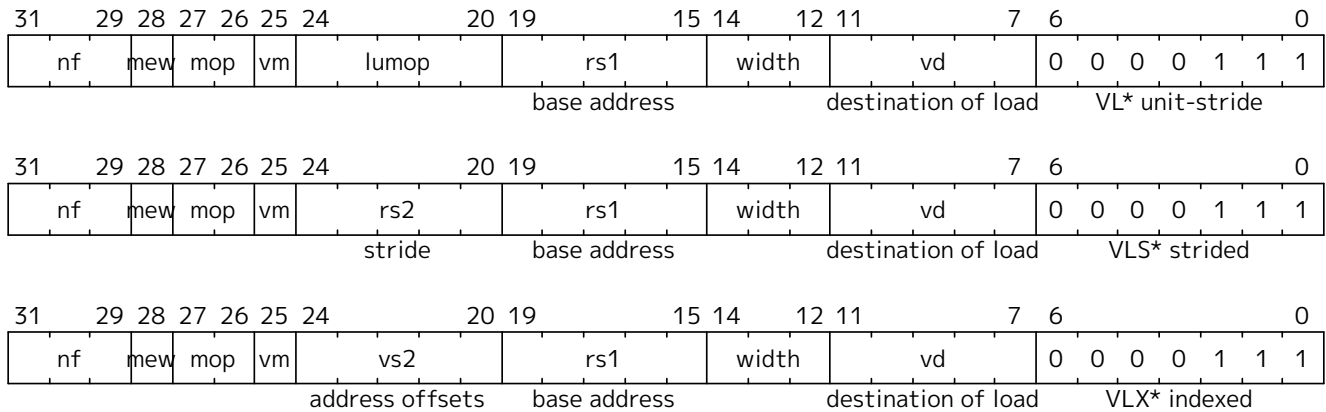
Vector loads and stores move values between vector registers and memory. Vector loads and stores can be masked, and they only access memory or raise exceptions for active elements. Masked vector loads do not update inactive elements in the destination vector register group, unless masked agnostic is specified (`vtype.vma=1`).

All vector loads and stores may generate and accept a non-zero `vstart` value.

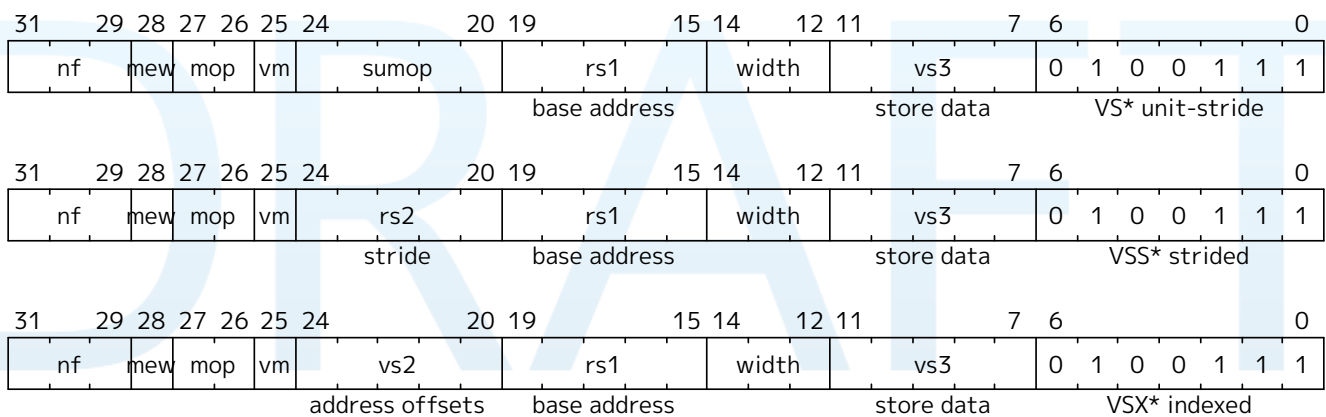
28.7.1. Vector Load/Store Instruction Encoding

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 28.5.3.1](#)).

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 28.7.3
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.

28.7.2. 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 28.7.8](#).

The **nf[2:0]** field also encodes the number of whole vector registers to transfer for the whole vector register load/store instructions.

28.7.3. 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.

	me w	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.

28.7.4. 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 $evl = \text{ceil}(vl/8)$ (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`.

28.7.5. 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.

28.7.6. 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 **eix** instead of **ex** 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.

28.7.7. 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.

28.7.8. 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 $EMUL * NFIELDS \leq 8$, otherwise the instruction encoding is reserved.



The product $\text{ceil}(EMUL) * NFIELDS$ 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.

28.7.8.1. 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.

28.7.8.2. 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.

28.7.8.3. 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 $EMUL * NFIELDs \leq 8$ 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 $x5+v5[i]+4$

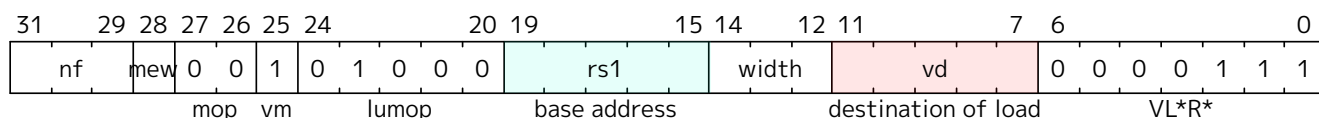
For vector indexed segment loads, the destination vector register groups cannot overlap the source vector register group (specified by **vs2**), else the instruction encoding is reserved.



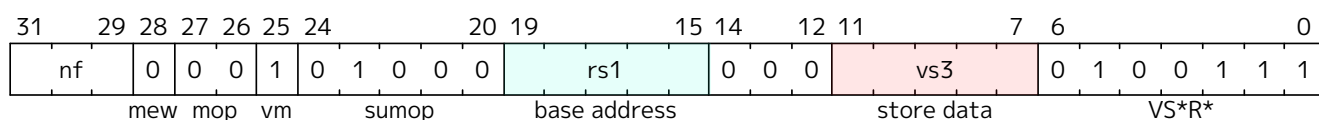
This constraint supports restart of indexed segment loads that raise exceptions partway through loading a structure.

28.7.9. 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** $\geq$ **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**

$\geq \text{evl}$.

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 ($\text{EEW}/8$) or the implementation's smallest supported SEW size in bytes ($\text{SEW}_{\text{MIN}}/8$).



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 $\text{SEW}_{\text{MIN}} > \text{XLEN}$ 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 $\text{VLEN}/8$ bytes held at address in $a0$

`vl1re16.v v3, (a0)` # Load $v3$ with $\text{VLEN}/16$ halfwords held at address in $a0$

`vl1re32.v v3, (a0)` # Load $v3$ with $\text{VLEN}/32$ words held at address in $a0$

`vl1re64.v v3, (a0)` # Load $v3$ with $\text{VLEN}/64$ doublewords held at address in $a0$

`vl2r.v v2, (a0)` # Pseudoinstruction equal to `vl2re8.v`

`vl2re8.v v2, (a0)` # Load $v2-v3$ with $2 \times \text{VLEN}/8$ bytes from address in $a0$

`vl2re16.v v2, (a0)` # Load $v2-v3$ with $2 \times \text{VLEN}/16$ halfwords held at address in $a0$

`vl2re32.v v2, (a0)` # Load $v2-v3$ with $2 \times \text{VLEN}/32$ words held at address in $a0$

`vl2re64.v v2, (a0)` # Load $v2-v3$ with $2 \times \text{VLEN}/64$ doublewords held at address in $a0$

`vl4r.v v4, (a0)` # Pseudoinstruction equal to `vl4re8.v`

`vl4re8.v v4, (a0)` # Load $v4-v7$ with $4 \times \text{VLEN}/8$ 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 $8 \times \text{VLEN}/8$ 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)
```

28.8. 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.

28.9. 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.

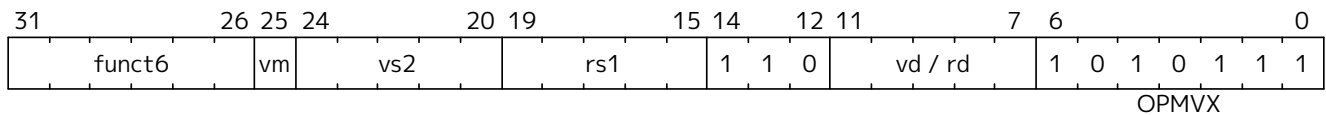
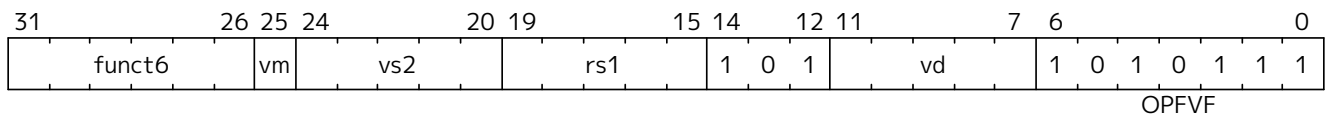
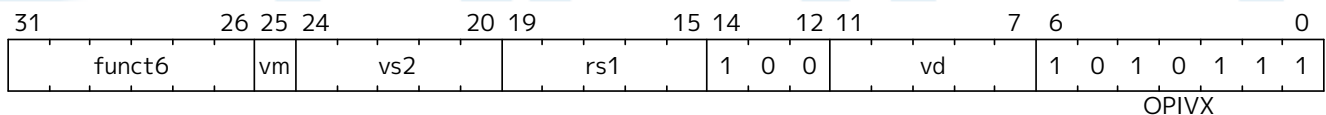
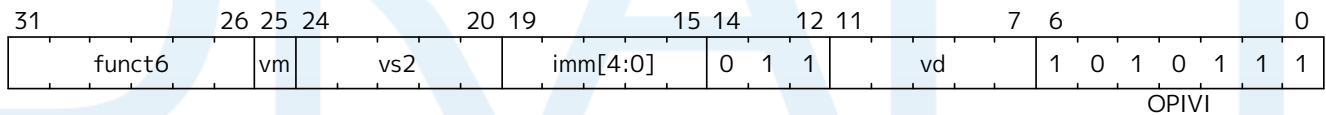
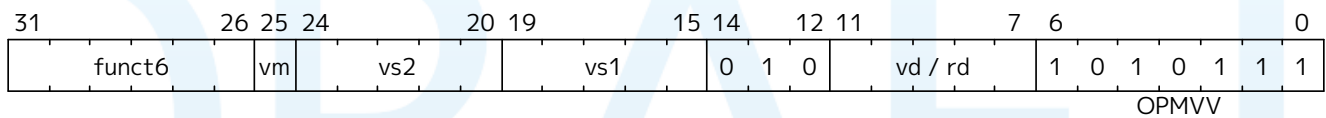
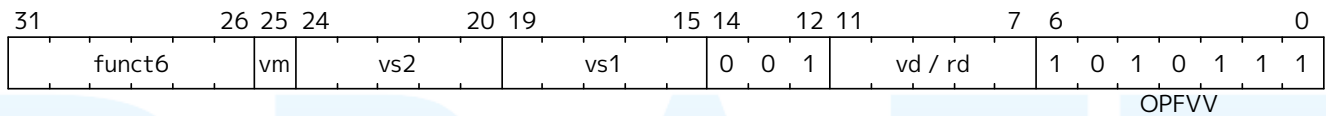
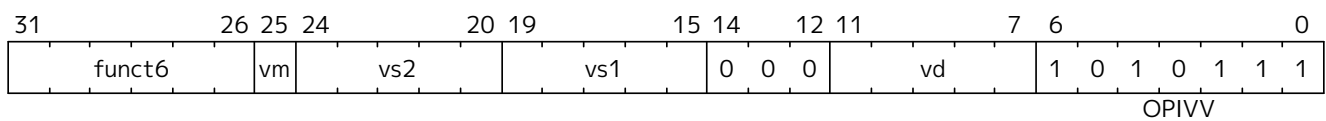


Treating the vector length and mask as control rather than data typically matches the semantics of the corresponding scalar code, where branch instructions ordinarily would have been used. Treating the mask as control allows masked vector load instructions to access memory before the mask value is known, without the need for a misspeculation-recovery mechanism.

28.10. Vector Arithmetic Instruction Formats

The vector arithmetic instructions use a new major opcode (OP-V = 101011₂) 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



28.10.1. 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.*

28.10.2. 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 **vwf*** 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 28.5.2](#).

28.10.3. 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 28.5.2](#).

28.11. Vector Integer Arithmetic Instructions

A set of vector integer arithmetic instructions is provided. Unless otherwise stated, integer operations wrap around on overflow.

28.11.1. 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
vrsb.vx vd, vs2, rs1, vm    # vd[i] = x[rs1] - vs2[i]
vrsb.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 = vrsb.vx vd,vs,x0` is provided.

28.11.2. 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
vwsb.vv vd, vs2, vs1, vm    # vector-vector
vwsb.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
vwsb.wv   vd, vs2, vs1, vm # vector-vector
vwsb.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.

28.11.3. 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
vsxt.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
vsxt.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
vsxt.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.

28.11.4. 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 28.4.5](#). 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.

vadc 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 **vmsbc** 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.

28.11.5. 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.
```

28.11.6. 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
```

28.11.7. 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**.*

28.11.8. 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 28.4.5](#). 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
<code>va < vb</code>	<code>vmslt{u}.vv vd, va, vb, vm</code>	
<code>va <= vb</code>	<code>vmsle{u}.vv vd, va, vb, vm</code>	
<code>va > vb</code>	<code>vmslt{u}.vv vd, vb, va, vm</code>	<code>vmsgt{u}.vv vd, va, vb, vm</code>
<code>va >= vb</code>	<code>vmsle{u}.vv vd, vb, va, vm</code>	<code>vmsge{u}.vv vd, va, vb, vm</code>
<code>va < x</code>	<code>vmslt{u}.vx vd, va, x, vm</code>	
<code>va <= x</code>	<code>vmsle{u}.vx vd, va, x, vm</code>	
<code>va > x</code>	<code>vmsgt{u}.vx vd, va, x, vm</code>	
<code>va >= x</code>	see below	
<code>va < i</code>	<code>vmsle{u}.vi vd, va, i-1, vm</code>	<code>vmslt{u}.vi vd, va, i, vm</code>
<code>va <= i</code>	<code>vmsle{u}.vi vd, va, i, vm</code>	
<code>va > i</code>	<code>vmsgt{u}.vi vd, va, i, vm</code>	
<code>va >= i</code>	<code>vmsgt{u}.vi vd, va, i-1, vm</code>	<code>vmsge{u}.vi vd, va, i, vm</code>
<code>va, vb</code> vector register groups		
<code>x</code> scalar integer register		
<code>i</code> 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).



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).



Because the 5-bit vector immediates are always sign-extended, when the high bit of the `sim5` immediate is set, `vmsleu.vi` and `vmsgtu.vi` also support unsigned immediate values in the range $2^{\text{SEW}}-16$ to $2^{\text{SEW}}-1$, allowing corresponding `vmsltu.vi` and `vmsgeu.vi` compares against unsigned immediates in the range $2^{\text{SEW}}-15$ to 2^{SEW} . Note that `vmsltu.vi` and `vmsgeu.vi` with immediate 2^{SEW} is not useful as it is always true or false, respectively.



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.

28.11.9. 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
```

28.11.10. 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 half of the product to the destination register, while the **mulh** versions write the high half 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.

28.11.11. 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.

28.11.12. 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
```

28.11.13. 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]
```

28.11.14. Vector Widening Integer Multiply-Add Instructions

The widening integer multiply-add instructions add the full $2 \times \text{SEW}$ -bit product from a SEW -bit $\times \text{SEW}$ -bit multiply to a $2 \times \text{SEW}$ -bit value and produce a $2 \times \text{SEW}$ -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]

28.11.15. 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]
```

28.11.16. 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.

28.12. 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 $-2^{N-1} \dots +2^{N-1}-1$, and unsigned integers in the range $0 \dots +2^N-1$. 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.

28.12.1. 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
```

28.12.2. 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)

```

28.12.3. Vector Single-Width Fractional Multiply with Rounding and Saturation

The signed fractional multiply instruction produces a $2 \times \text{SEW}$ 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`).

28.12.4. 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 $\lg 2(\text{SEW})$

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)
```

28.12.5. 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 $\lg_2(2*SEW)$ 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.vv 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.vv 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.

28.13. 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 `0ff` 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 `0ff` 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 28.10.1](#).

28.13.1. 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.

28.13.2. 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
vfsb.vv vd, vs2, vs1, vm    # Vector-vector
vfsb.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]
```

28.13.3. 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
vfwsb.vv vd, vs2, vs1, vm # vector-vector
vfwsb.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
vfwsb.wv vd, vs2, vs1, vm # vector-vector
vfwsb.wf vd, vs2, rs1, vm # vector-scalar
```

28.13.4. 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]
```

28.13.5. 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
```

28.13.6. 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
vfmac.vv vd, vs1, vs2, vm # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vfmac.vf vd, rs1, vs2, vm # vd[i] = (f[rs1] * vs2[i]) + vd[i]

# FP negate-(multiply-accumulate), overwrites subtrahend
vfnmac.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
vfmsac.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vd[i]) + vs2[i]
vfmsac.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vd[i]) + vs2[i]

# FP negate-(multiply-add), overwrites multiplicand
vfnmsac.vv vd, vs1, vs2, vm    # vd[i] = -(vs1[i] * vd[i]) - vs2[i]
vfnmsac.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.

28.13.7. 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
vfwmac.vv vd, vs1, vs2, vm    # vd[i] = (vs1[i] * vs2[i]) + vd[i]
vfwmac.vf vd, rs1, vs2, vm    # vd[i] = (f[rs1] * vs2[i]) + vd[i]

# FP widening negate-(multiply-accumulate), overwrites addend
vfnwmac.vv vd, vs1, vs2, vm    # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfnwmac.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
vfnwmsac.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]
```

28.13.8. 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
```

28.13.9. 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 **vfrsqрте7** but this was deemed to cause confusion with the **ex** 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
$-\infty \leq x < -0.0$	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 $\text{floor}((3*B -$

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. *vfrsqt7.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 $SEW=32$, $vfrsqr7(0x00718abc \approx 1.043e-38) = 0x5f080000 \approx 9.800e18$, and $vfrsqr7(0x7f765432 \approx 3.274e38) = 0x1f820000 \approx 5.506e-20$.



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.

28.13.10. 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 **ex** 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	
$-2^{B+1} < x \leq -2^B$ (normal)	any	$-2^{-(B+1)} \geq y > -2^{-B}$ (subnormal, sig=01...)	
$-2^B < x \leq -2^{B-1}$ (normal)	any	$-2^{-B} \geq y > -2^{-(B+1)}$ (subnormal, sig=1...)	
$-2^{B-1} < x \leq -2^{-(B+1)}$ (normal)	any	$-2^{-(B+1)} \geq y > -2^{B-1}$ (normal)	

Input (x)	Rounding Mode	Output ($y \approx 1/x$)	Exceptions raised
$-2^{-(B+1)} < x \leq -2^{-B}$ (subnormal, sig=1...)	any	$-2^{B-1} \geq y > -2^B$ (normal)	
$-2^{-B} < x \leq -2^{-(B+1)}$ (subnormal, sig=01...)	any	$-2^B \geq y > -2^{B+1}$ (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
$2^{-(B+1)} \leq x < 2^{-B}$ (subnormal, sig=01...)	any	$2^{B+1} > y \geq 2^B$ (normal)	
$2^{-B} \leq x < 2^{-(B+1)}$ (subnormal, sig=1...)	any	$2^B > y \geq 2^{B-1}$ (normal)	
$2^{B+1} \leq x < 2^{B-1}$ (normal)	any	$2^{B-1} > y \geq 2^{B+1}$ (normal)	
$2^{B-1} \leq x < 2^B$ (normal)	any	$2^{B+1} > y \geq 2^{-B}$ (subnormal, sig=1...)	
$2^B \leq x < 2^{B+1}$ (normal)	any	$2^{-B} > y \geq 2^{-(B+1)}$ (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 $(2*B - 1 - \text{the normalized input exponent})$. 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. vfred7.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 $SEW=32$, $vfrec7(0x00718abc \approx 1.043e-38) = 0x7e900000 \approx 9.570e37$, and $vfrec7(0x7f765432 \approx 3.274e38) = 0x00214000 \approx 3.053e-39$.



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.

28.13.11. 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
```

28.13.12. 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: `vneg.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`.

28.13.13. 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 28.4.5](#). 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
<code>va < vb</code>	<code>vmflt.vv vd, va, vb, vm</code>	
<code>va <= vb</code>	<code>vmfle.vv vd, va, vb, vm</code>	
<code>va > vb</code>	<code>vmflt.vv vd, vb, va, vm</code>	<code>vmfgt.vv vd, va, vb, vm</code>
<code>va >= vb</code>	<code>vmfle.vv vd, vb, va, vm</code>	<code>vmfge.vv vd, va, vb, vm</code>
<code>va < f</code>	<code>vmflt.vf vd, va, f, vm</code>	
<code>va <= f</code>	<code>vmfle.vf vd, va, f, vm</code>	
<code>va > f</code>	<code>vmfgt.vf vd, va, f, vm</code>	
<code>va >= f</code>	<code>vmfge.vf vd, va, f, vm</code>	
<code>va, vb</code>	vector register groups	
<code>f</code>	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.

28.13.14. 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.

28.13.15. 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]
```

28.13.16. 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`.

28.13.17. 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.

28.13.18. 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 28.10.2](#)).



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.

28.13.19. 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 28.10.3](#)).



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.

28.14. 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 ($0 < \text{index} < \text{VLEN}/\text{SEW}$) are considered the tail and are managed with the current tail agnostic/undisturbed policy.

If $\text{vl}=0$, no operation is performed and the destination register is not updated.



This choice of behavior for $\text{vl}=0$ 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 $\text{vl}=0$. 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 $\text{vl}=0$ will have to be added to ensure that the source scalar is copied to the destination (e.g., by explicitly setting $\text{vl}=1$ 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.

28.14.1. 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[*] )
```

28.14.2. 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))
```

28.14.3. 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`.

28.14.3.1. 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:

$$vd[0] = (((vs1[0] + vs2[0]) + vs2[1]) + \dots) + vs2[vl-1]$$

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.*

28.14.3.2. 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 $-\infty$) 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.*

28.14.3.3. 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.*

28.14.4. 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 $2 \times \text{SEW}$ bits before adding to the $2 \times \text{SEW}$ -bit accumulator.



`vfwredosum.vs` handles inactive elements and NaN payloads analogously to `vfredosum.vs`; `vfwredusum.vs` does so analogously to `vfredusum.vs`.

28.15. Vector Mask Instructions

Several instructions are provided to help operate on mask values held in a vector register.

28.15.1. 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.

28.15.2. 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 28.4.5](#).

The **vpopc.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.

```
vpopc.m rd, vs2, v0.t # x[rd] = sum_i ( vs2.mask[i] && v0.mask[i] )
```

The **vpopc.m** instruction writes **x[rd]** even if **vl=0** (with the value 0, since no mask elements are active).

Traps on **vpopc.m** are always reported with a **vstart** of 0. The **vpopc.m** instruction will raise an illegal-instruction exception if **vstart** is non-zero.

28.15.3. 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.

28.15.4. 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**).

28.15.5. 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**).

28.15.6. **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**).

28.15.7. 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

```

28.15.8. 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

```

28.15.9. 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.*

28.16. Vector Permutation Instructions

A range of permutation instructions are provided to move elements around within the vector registers.

28.16.1. 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 $vstart \geq vl$ 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 $vstart \geq vl$, 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.

28.16.2. 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 $vstart \geq vl$ 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 $vstart \geq vl$, 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.

28.16.3. 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 **vsliedown**. In particular, power-of-2 offsets may operate substantially faster than other offsets.*

For all of the **vslideup**, **vsliedown**, **v[f]slide1up**, and **v[f]slide1down** instructions, if **vstart** $\geq$ **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.

28.16.3.1. 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

$0 \leq i < \min(\text{vl}, \max(\text{vstart}, \text{OFFSET}))$	Unchanged
$\max(\text{vstart}, \text{OFFSET}) \leq i < \text{vl}$	$\text{vd}[i] = \text{vs2}[i - \text{OFFSET}]$
if v0.mask[i] enabled	
$\text{vl} \leq i < \text{VLMAX}$	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**.*

28.16.3.2. Vector Slide-down Instructions

```
vsliedown.vx vd, vs2, rs1, vm     # vd[i] = vs2[i+x[rs1]]
vsliedown.vi vd, vs2, uimm, vm    # vd[i] = vs2[i+uimm]
```

For **vsliedown**, 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 28.3.4.3](#)).

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**)

$0 \leq i + \text{OFFSET} < \text{VLMAX}$	$\text{src}[i] = \text{vs2}[i + \text{OFFSET}]$
$\text{VLMAX} \leq i + \text{OFFSET}$	$\text{src}[i] = 0$

vslidedown behavior for destination element *i* in slide (**vstart** < **vl**)

$0 \leq i < \text{vstart}$	Unchanged
$\text{vstart} \leq i < \text{vl}$	$\text{vd}[i] = \text{src}[i]$ if v0.mask[i] enabled
$\text{vl} \leq i < \text{VLMAX}$	Follow tail policy

28.16.3.3. 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 28.3.4.3](#)).

vslide1up behavior when **vl** > 0

$i < \text{vstart}$	unchanged
$0 = i = \text{vstart}$	$\text{vd}[i] = \text{x}[\text{rs1}]$ if v0.mask[i] enabled
$\max(\text{vstart}, 1) \leq i < \text{vl}$	$\text{vd}[i] = \text{vs2}[i-1]$ if v0.mask[i] enabled
$\text{vl} \leq i < \text{VLMAX}$	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.

28.16.3.4. 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.

28.16.3.5. 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 28.3.4.3](#)).

```
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.

28.16.3.6. 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.

28.16.4. 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 28.3.4.3](#)). 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.

28.16.5. 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.vv 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 28.3.4.3](#)).

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.vv v2, v1, v0
1 2 3 4 8 7 5 2 0	v2

The `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 to do that but will instead restart from element 0. This does mean elements in destination register after `vstart` will already have been updated.

28.16.5.1. 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
```

28.16.6. 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** $\geq$ **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** $\geq$ **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 **sim** field (**sim**[2:0]) using the same encoding as the **nf**[2:0] field for memory instructions (Figure Table 55), i.e., **sim**[2:0] = NREG-1.

The value of NREG must be 1, 2, 4, or 8, and values of **sim**[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.

28.17. 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.*

28.17.1. 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**.

28.17.2. 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.

28.17.3. 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.

28.17.4. 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.

28.18. 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.

28.18.1. 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.

28.18.2. 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 28.6](#)).

All Zve* extensions support all vector load and store instructions ([Section 28.7](#)), except Zve64* extensions do not support EEW=64 for index values when XLEN=32.

All Zve* extensions support all vector integer instructions ([Section 28.11](#)), except that the `vmulh` integer multiply variants that return the high half 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 28.12](#)), 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 28.14.1](#), [Section 28.14.2](#)).

All Zve* extensions support all vector mask instructions ([Section 28.15](#)).

All Zve* extensions support all vector permutation instructions ([Section 28.16](#)), 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 28.13](#)) for floating-point operands with EEW=32. Vector single-width floating-point reduction operations ([Section 28.14.3](#)) for EEW=32 are supported.

The Zve64d extension depends upon the D extension, and implements all vector floating-point

instructions (Section 28.13) 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 28.14.3) for EEW=32 and EEW=64 are supported as well as widening reductions from FP32 to FP64.

28.18.3. V: Vector Extension for Application Processors

The single-letter V extension is intended for use in application processor profiles.

The `misae.v` bit is set for implementations providing `misae` 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 28.6).

The V extension supports all vector load and store instructions (Section 28.7), except the V extension does not support EEW=64 for index values when XLEN=32.

The V extension supports all vector integer instructions (Section 28.11).

The V extension supports all vector fixed-point arithmetic instructions (Section 28.12).

The V extension supports all vector integer single-width and widening reduction operations (Section 28.14.1, Section 28.14.2).

The V extension supports all vector mask instructions (Section 28.15).

The V extension supports all vector permutation instructions (Section 28.16).

The V extension depends upon the F and D extensions, and implements all vector floating-point instructions (Section 28.13) 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 28.14.3) 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.

28.18.4. 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.

28.18.5. 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 28.13](#), [Section 28.14.3](#), [Section 28.14.4](#), [Section 28.16.2](#), [Section 28.16.3.4](#), and [Section 28.16.3.6](#) 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 (`vfnvct[.rtz].x[u].f.w`) and integer-to-floating-point widening conversions (`vfwvct.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.

28.19. 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.

28.19.1. 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).

28.19.2. 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 28.6.3](#)). 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.

28.19.3. 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.

28.19.4. 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.

28.19.5. 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 Zvksha 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.

28.19.6. 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).

28.20. Vector Instruction Listing

Integer				Integer				FP			
funct3				funct3				funct3			
OPIVV	V			OPMV	V			OPFVV	V		
OPIVX		X		OPMV		X		OPFVF		F	
OPIVI			I								

funct6					funct6				funct6			
00000 0	V	X	I	vadd	00000 0	V		vredsu m	00000 0	V	F	vfadd
00000 1					00000 1	V		vredan d	00000 1	V		vfredus um
00001 0	V	X		vsub	00001 0	V		vredor	00001 0	V	F	vfsub
00001 1		X	I	vrsb	00001 1	V		vredxor	00001 1	V		vfredos um
00010 0	V	X		vminu	00010 0	V		vredmi nu	00010 0	V	F	vfmin
00010 1	V	X		vmin	00010 1	V		vredmi n	00010 1	V		vfredm in
00011 0	V	X		vmaxu	00011 0	V		vredma xu	00011 0	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			vrgatheri16								
001111		X	I	vslidedown	001111		X	vslide1down	001111		F	vfslide1down

funct6					funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXUNARYO	010000	V		VWFUNARYO
					010000		X	VRXUNARYO	010000		F	VRFUNARYO
010001	V	X	I	vmadc	010001				010001			
010010	V	X		vsbc	010010	V		VXUNARYO	010010	V		VFUNARYO
010011	V	X		vmsbc	010011				010011	V		VFUNARY1
010100					010100	V		VMUNARYO	010100			
010101					010101				010101			
010110					010110				010110			
010111	V	X	I	vmerge/vmv	010111	V		vcompress	010111		F	vmerge/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	vfddiv
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<n r>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	vfmzac
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		vfwredsum
110010					110010	V	X	vwsubu	110010	V	F	vfwsub
110011					110011	V	X	vwsu	110011	V		vfwredsum
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	vwmul u	111000	V	F	vfwmul
111001					111001				111001			
111010					111010	V	X	vwmuls u	111010			
111011					111011	V	X	vwmul	111011			
111100					111100	V	X	vwmac cu	111100	V	F	vfwmac c
111101					111101	V	X	vwmac c	111101	V	F	vfwnm acc
111110					111110		X	vwmac cus	111110	V	F	vfwm sa c
111111					111111	V	X	vwmac csu	111111	V	F	vfwnm sac

DRAFT

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 29. Cryptography Extensions: Scalar & Entropy Source Instructions, Version 1.0.1

29.1. Introduction

This chapter 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 chapter also describes the architectural interface to an Entropy Source, which can be used to generate cryptographic secrets. This is found in [Section 29.4](#).

It also contains a mechanism allowing core implementers to provide "*Constant Time Execution*" guarantees in [Section 29.5](#).

29.1.1. 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 29.5](#)) and the entropy source ([Section 29.4](#)) 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.

29.1.2. 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 29.9](#). The [Sail Manual](#) is recommended reading in order to best understand the code snippets.

Note that this chapter contains only a subset of the formal model: refer to the formal model [GitHub repository](#) for the complete model.

29.1.3. 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.

29.2. 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.

29.2.1. 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](#).

29.2.2. 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](#).

29.2.3. 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](#).

29.2.4. Zknd - NIST Suite: AES Decryption

Instructions for accelerating the decryption and key-schedule functions of the AES block cipher.

RV32	RV64	Mnemonic	Instruction
✓		aes32dsi	Section 29.3.1
✓		aes32dsmi	Section 29.3.2
	✓	aes64ds	Section 29.3.5
	✓	aes64dsm	Section 29.3.6
	✓	aes64im	Section 29.3.9
	✓	aes64ksi	Section 29.3.10

RV32	RV64	Mnemonic	Instruction
	✓	aes64ks2	Section 29.3.11

NOTE: The [Section 29.3.10](#) and [Section 29.3.11](#) instructions are present in both the [Section 29.2.4](#) and [Section 29.2.5](#) extensions.

29.2.5. zkne - NIST Suite: AES Encryption

Instructions for accelerating the encryption and key-schedule functions of the AES block cipher.

RV32	RV64	Mnemonic	Instruction
✓		aes32esi	Section 29.3.3
✓		aes32esmi	Section 29.3.4
	✓	aes64es	Section 29.3.7
	✓	aes64esm	Section 29.3.8
	✓	aes64ks1i	Section 29.3.10
	✓	aes64ks2	Section 29.3.11

NOTE: The **aes64ks1i** and **aes64ks2** instructions are present in both the [Section 29.2.4](#) and [Section 29.2.5](#) extensions.

29.2.6. 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	Section 29.3.27
✓	✓	sha256sig1	Section 29.3.28
✓	✓	sha256sum0	Section 29.3.29
✓	✓	sha256sum1	Section 29.3.30
✓		sha512sig0h	Section 29.3.31
✓		sha512sig0l	Section 29.3.32
✓		sha512sig1h	Section 29.3.33
✓		sha512sig1l	Section 29.3.34
✓		sha512sum0r	Section 29.3.35
✓		sha512sum1r	Section 29.3.36
	✓	sha512sig0	Section 29.3.37
	✓	sha512sig1	Section 29.3.38
	✓	sha512sum0	Section 29.3.39
	✓	sha512sum1	Section 29.3.40

29.2.7. 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	Section 29.3.43
✓	✓	sm4ks	Section 29.3.44

29.2.8. zksh - ShangMi Suite: SM3 Hash Function Instructions

Instructions for accelerating the SM3 hash function.

RV32	RV64	Mnemonic	Instruction
✓	✓	sm3p0	Section 29.3.41
✓	✓	sm3p1	Section 29.3.42

29.2.9. 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 29.4](#) for the normative specification and access control notes. [Section 29.7](#) contains design rationale and further recommendations to implementers.

29.2.10. zkn - NIST Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Section 29.2.1	Bitmanipulation instructions for cryptography.
Section 29.2.2	Carry-less multiply instructions.
Section 29.2.3	Cross-bar Permutation instructions.
Section 29.2.5	AES encryption instructions.
Section 29.2.4	AES decryption instructions.
Section 29.2.6	SHA2 hash function instructions.

A core which implements **Zkn** must implement all of the above extensions.

29.2.11. zks - ShangMi Algorithm Suite

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Section 29.2.1	Bitmanipulation instructions for cryptography.
Section 29.2.2	Carry-less multiply instructions.
Section 29.2.3	Cross-bar Permutation instructions.
Section 29.2.7	SM4 block cipher instructions.
Section 29.2.8	SM3 hash function instructions.

A core which implements **Zks** must implement all of the above extensions.

29.2.12. Zk - Standard scalar cryptography extension

This extension is shorthand for the following set of other extensions:

Included Extension	Description
Section 29.2.10	NIST Algorithm suite extension.
Section 29.2.9	Entropy Source extension.
Zkt	Data independent execution latency extension.

A core which implements **Zk** must implement all of the above extensions.

29.2.13. 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 29.5](#).

29.3. Instructions

29.3.1. aes32dsi

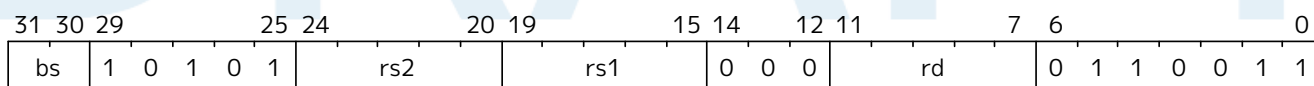
Synopsis

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
Section 29.2.4 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

DRAFT

29.3.2. aes32dsmi

Synopsis

AES middle round decryption instruction for RV32.

Mnemonic

aes32dsmi rd, rs1, rs2, bs

Encoding

31	30	29				25	24					20	19					15	14			12	11					7	6					0
bs		1	0	1	1	1	rs2			rs1			0			0	0	rd			0			1	1	0	0	1	1					

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
Section 29.2.4 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.3. aes32esi

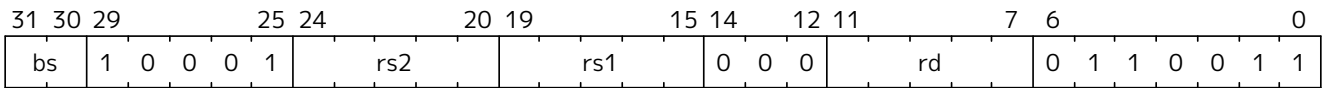
Synopsis

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
Section 29.2.5 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.4. aes32esmi

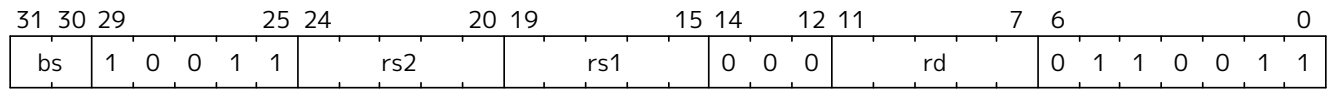
Synopsis

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) = EXT8(result); RETIRE_SUCCESS
}
```

Included in

Extension	Minimum version	Lifecycle state
Section 29.2.5 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.5. aes64ds

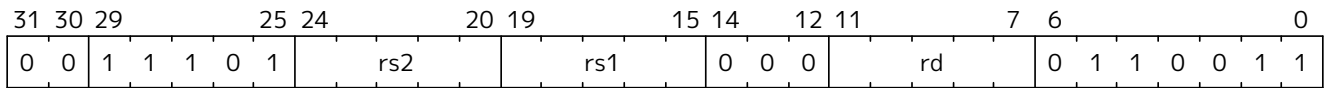
Synopsis

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
Section 29.2.4 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.6. aes64dsm

Synopsis

AES middle round decryption instruction for RV64.

Mnemonic

aes64dsm rd, rs1, rs2

Encoding

31	30	29				25	24				20	19				15	14	12	11				7	6				0
0	0	1	1	1	1	rs2			rs1			0			0	0	rd			0			1	1	0	0	1	1

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
Section 29.2.4 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.7. aes64es

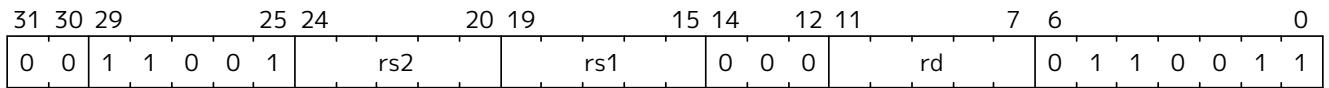
Synopsis

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_shiftrrows_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
Section 29.2.5 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.8. aes64esm

Synopsis

AES middle round encryption instruction for RV64.

Mnemonic

aes64esm rd, rs1, rs2

Encoding

31	30	29				25	24				20	19				15	14	12	11				7	6				0	
0	0	1	1	0	1	1	rs2			rs1			0			0	0	rd			0			1	1	0	0	1	1

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
Section 29.2.5 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.9. aes64im

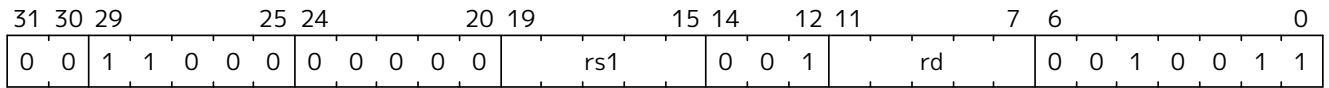
Synopsis

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
Section 29.2.4 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.10. aes64ksli

Synopsis

This instruction implements part of the KeySchedule operation for the AES Block cipher involving the SBox operation.

Mnemonic

aes64ksli rd, rs1, rnum

Encoding

31	30	29		25	24	23		20	19		15	14		12	11		7	6									0		
0	0	1	1	0	0	0	1	rnum				rs1				0	0	1	rd				0	0	1	0	0	1	1

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 `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
Section 29.2.5 (RV64)	v1.0.0	Ratified
Section 29.2.4 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.11. aes64ks2

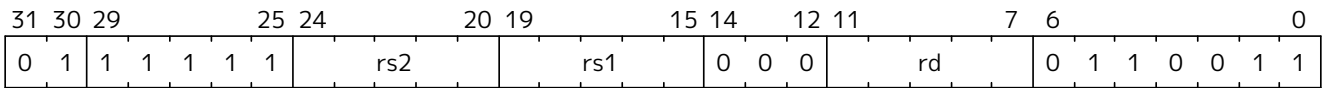
Synopsis

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
Section 29.2.5 (RV64)	v1.0.0	Ratified
Section 29.2.4 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.12. andn

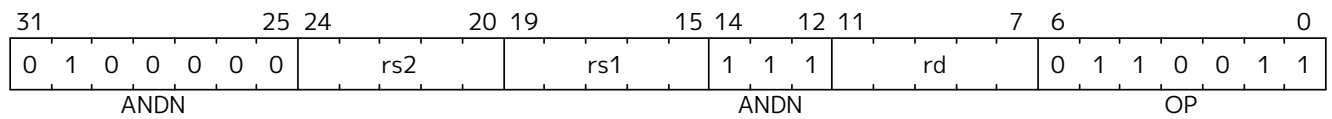
Synopsis

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

29.3.13. brev8

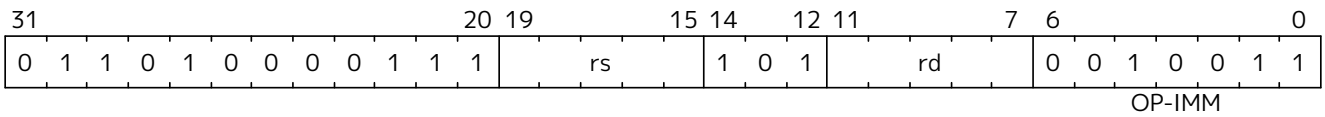
Synopsis

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

29.3.14. clmul

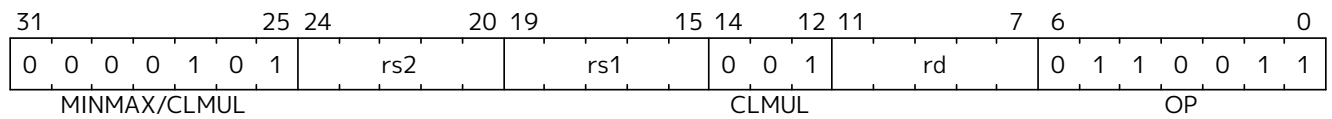
Synopsis

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

29.3.15. clmulh

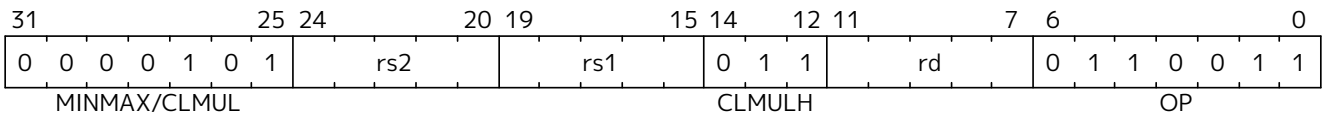
Synopsis

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

29.3.16. orn

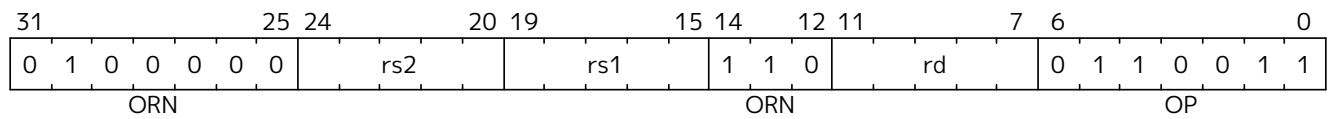
Synopsis

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

29.3.17. pack

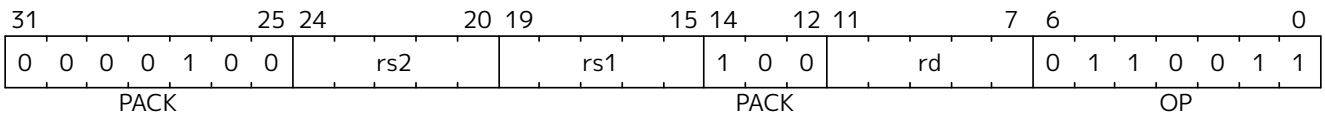
Synopsis

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

29.3.18. packh

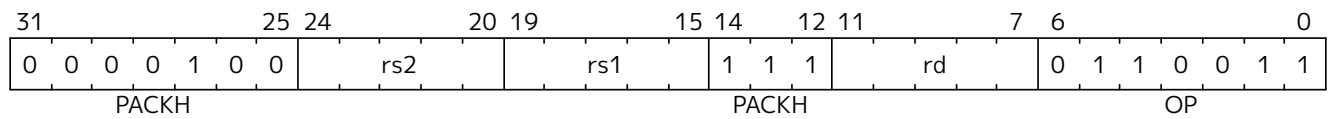
Synopsis

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

29.3.19. packw

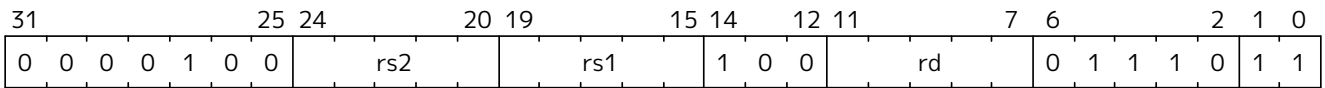
Synopsis

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

29.3.20. rev8

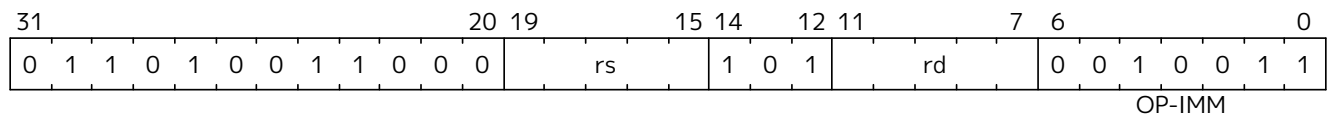
Synopsis

Byte-reverse register

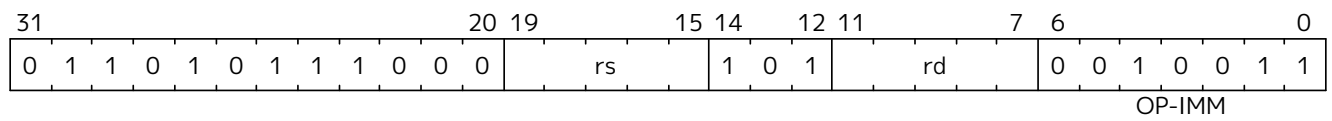
Mnemonic

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

29.3.21. rol

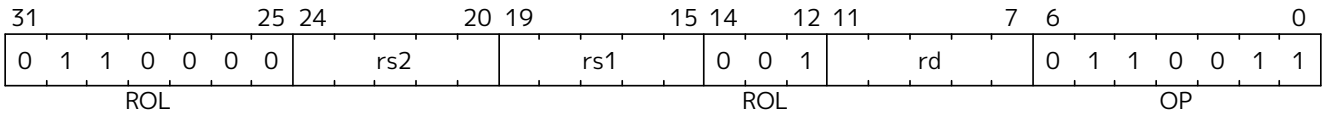
Synopsis

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

29.3.22. rolw

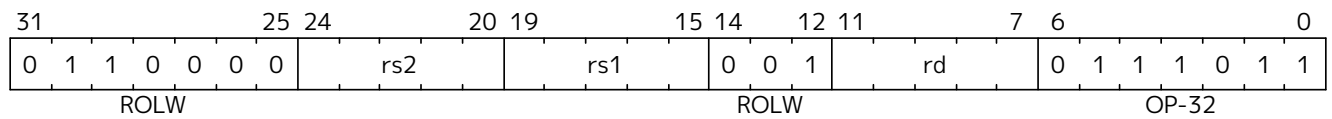
Synopsis

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

29.3.23. ror

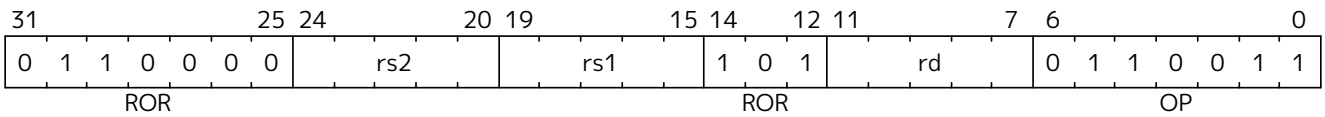
Synopsis

Rotate Right

Mnemonic

ror *rd*, *rs1*, *rs2*

Encoding



Description

This instruction performs a rotate right of *rs1* by the amount in least-significant $\log_2(\text{XLEN})$ 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

29.3.24. rori

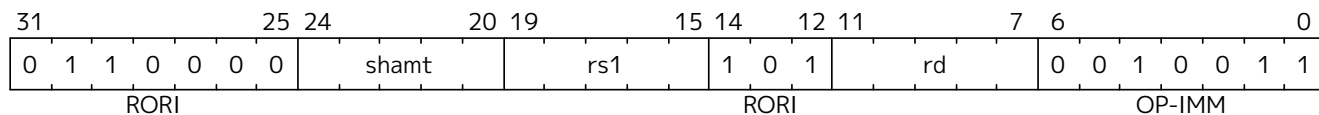
Synopsis

Rotate Right (Immediate)

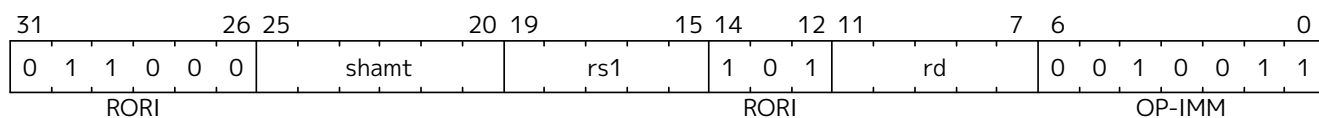
Mnemonic

rori *rd, rs1, shamt*

Encoding (RV32)



Encoding (RV64)



Description

This instruction performs a rotate right of *rs1* by the amount in the least-significant $\log_2(\text{XLEN})$ 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

29.3.25. roriw

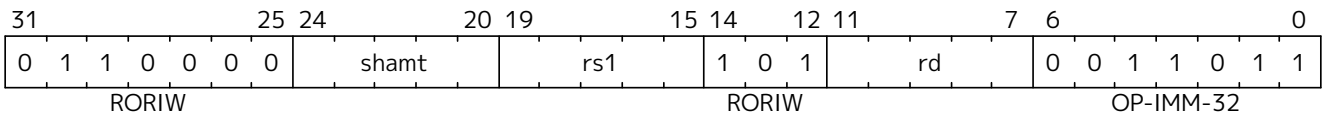
Synopsis

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 $\log_2(\text{XLEN})$ 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) = EXTZ(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

29.3.27. sha256sig0

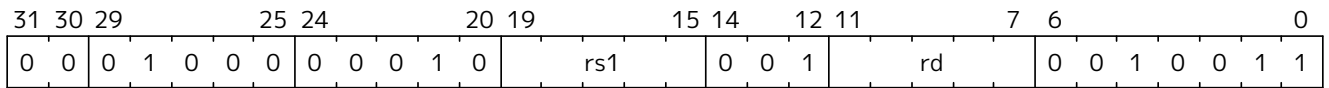
Synopsis

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
Section 29.2.6	v1.0.0	Ratified
Section 29.2.10	v1.0.0	Ratified
Section 29.2.12	v1.0.0	Ratified

29.3.28. sha256sig1

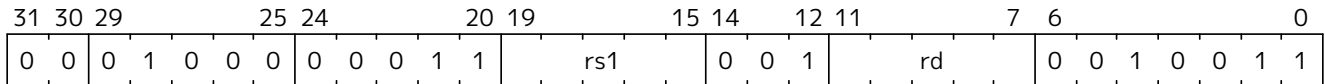
Synopsis

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
Section 29.2.6	v1.0.0	Ratified
Section 29.2.10	v1.0.0	Ratified
Section 29.2.12	v1.0.0	Ratified

29.3.29. sha256sum0

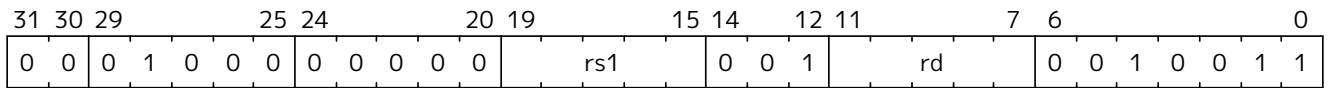
Synopsis

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
Section 29.2.6	v1.0.0	Ratified
Section 29.2.10	v1.0.0	Ratified
Section 29.2.12	v1.0.0	Ratified

29.3.30. sha256sum1

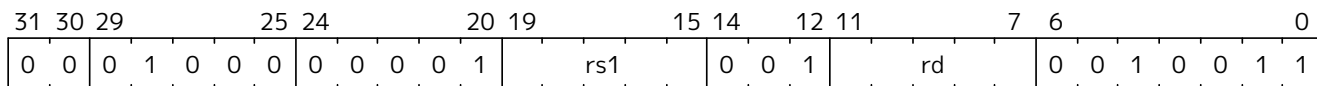
Synopsis

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
Section 29.2.6	v1.0.0	Ratified
Section 29.2.10	v1.0.0	Ratified
Section 29.2.12	v1.0.0	Ratified

29.3.31. sha512sig0h

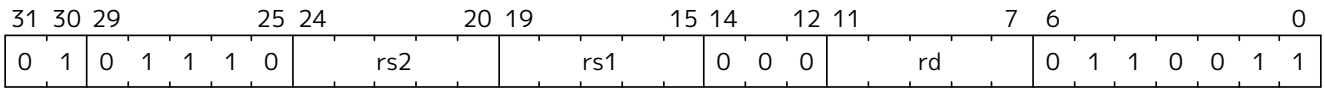
Synopsis

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.32. sha512sig0l

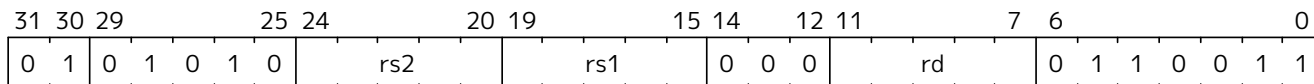
Synopsis

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.33. sha512sig1h

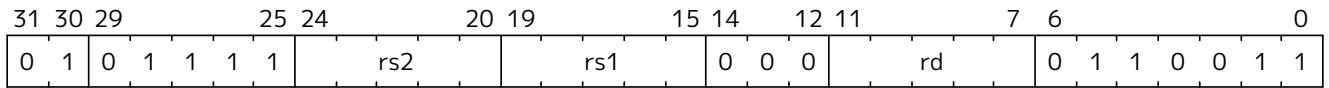
Synopsis

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.34. sha512sig1l

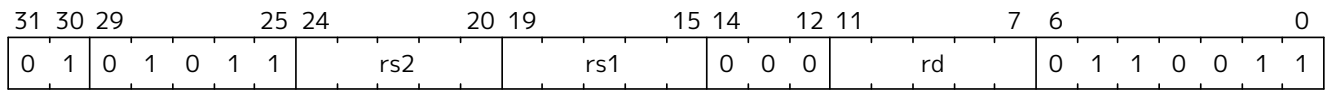
Synopsis

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.35. sha512sum0r

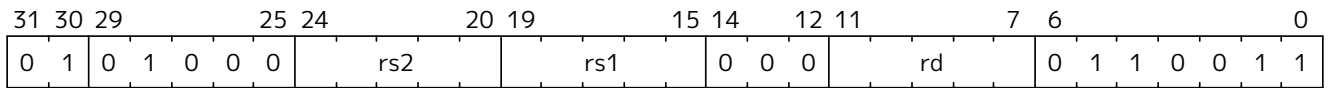
Synopsis

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.36. sha512sum1r

Synopsis

Implements the Sum1 transformation, as used in the SHA2-512 hash function ([NIST, 2015](#)).

Mnemonic

sha512sum1r rd, rs1, rs2

Encoding

31	30	29				25	24				20	19				15	14	12	11				7	6				0	
0	1	0	1	0	0	1	rs2			rs1			0			0	0	rd			0			1	1	0	0	1	1

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
Section 29.2.6 (RV32)	v1.0.0	Ratified
Section 29.2.10 (RV32)	v1.0.0	Ratified
Section 29.2.12 (RV32)	v1.0.0	Ratified

29.3.37. sha512sig0

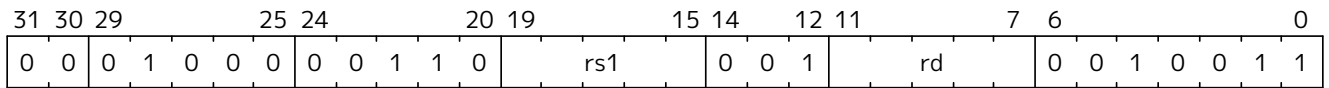
Synopsis

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
Section 29.2.6 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.38. sha512sig1

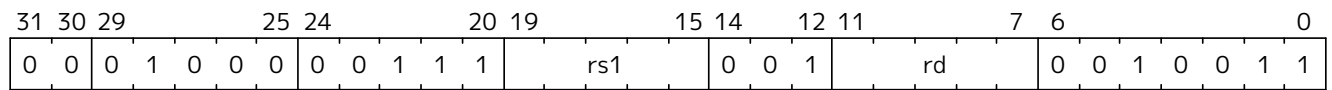
Synopsis

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
Section 29.2.6 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.39. sha512sum0

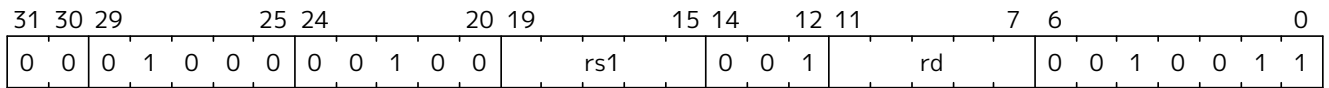
Synopsis

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
Section 29.2.6 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.40. sha512sum1

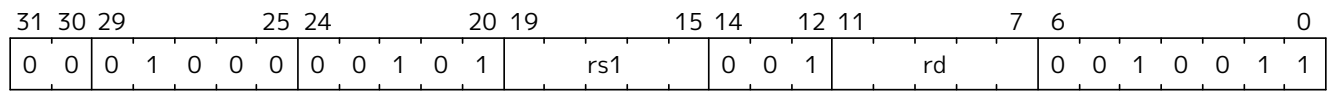
Synopsis

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
Section 29.2.6 (RV64)	v1.0.0	Ratified
Section 29.2.10 (RV64)	v1.0.0	Ratified
Section 29.2.12 (RV64)	v1.0.0	Ratified

29.3.41. sm3p0

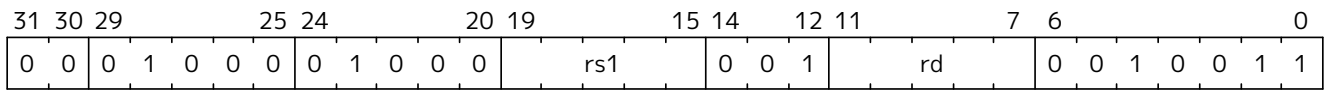
Synopsis

Implements the *PO* 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 *PO* 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
Section 29.2.8	v1.0.0	Ratified
Section 29.2.11	v1.0.0	Ratified

29.3.42. sm3p1

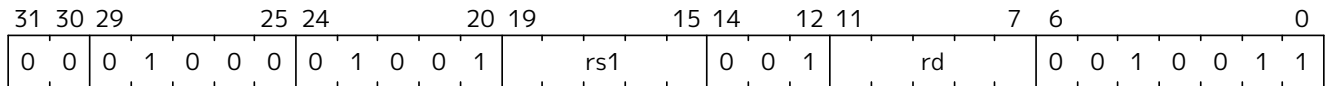
Synopsis

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
Section 29.2.8	v1.0.0	Ratified
Section 29.2.11	v1.0.0	Ratified

29.3.43. sm4ed

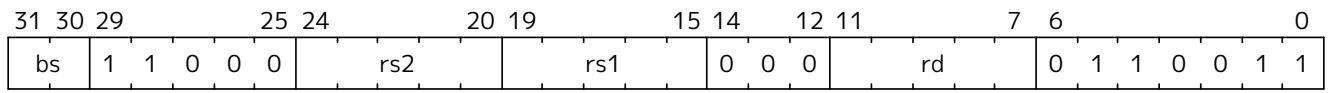
Synopsis

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
Section 29.2.7	v1.0.0	Ratified
Section 29.2.11	v1.0.0	Ratified

29.3.44. 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

31 30 29			25 24			20 19			15 14			12 11		7 6		0	
bs			1 1 0 1 0			rs2			rs1			0 0 0		rd		0 1 1 0 0 1 1	

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
Section 29.2.7	v1.0.0	Ratified
Section 29.2.11	v1.0.0	Ratified

29.3.45. unzip

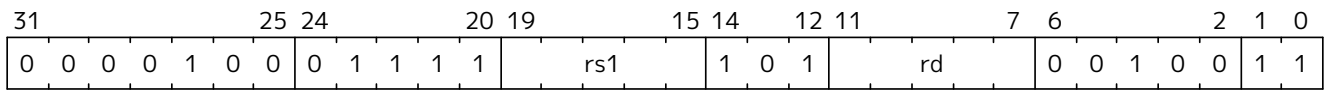
Synopsis

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

29.3.46. xnor

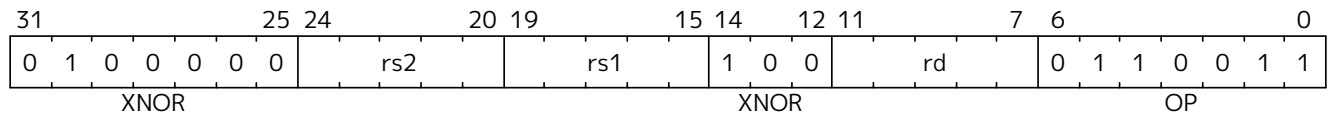
Synopsis

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*) = $\sim(X(rs1) \wedge X(rs2))$;

Included in

Extension	Minimum version	Lifecycle state
Zbb (Basic bit-manipulation)	v1.0.0	Ratified
Zbkb (Zbkb)	v1.0.0-rc4	Ratified

29.3.47. xperm8

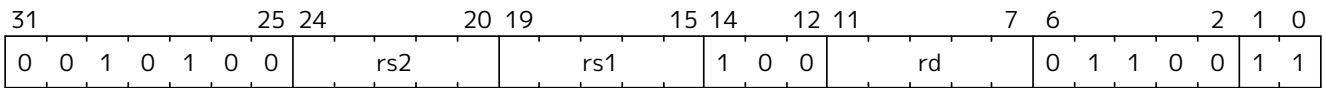
Synopsis

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

29.3.48. xperm4

Synopsis

Nibble-wise lookup of indices into a vector.

Mnemonic

xperm4 *rd*, *rs1*, *rs2*

Encoding

31		25	24		20	19		15	14	12	11		7	6		2	1	0
0	0	1	0	1	0	0												
							rs2					rs1						
										0	1	0			rd			
																0	1	1
																0	0	1
																		1

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

29.3.49. zip

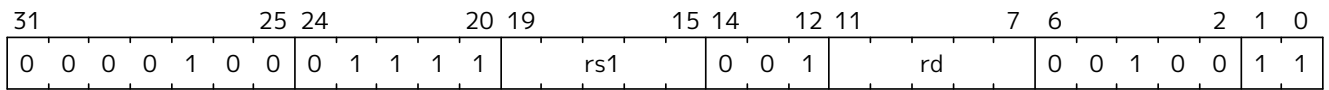
Synopsis

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

29.4. 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 29.7 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.

29.4.1. The seed CSR

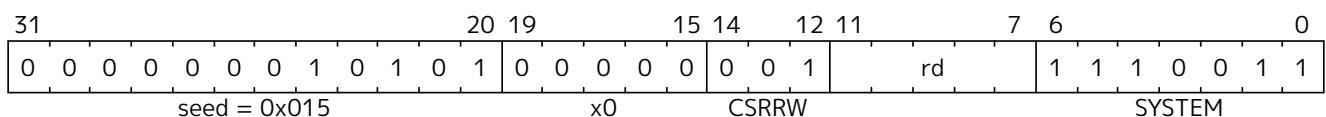
seed is an unprivileged CSR located at address **0x015**. The 32-bit contents of **seed** are as follows:

Bits	Name	Description
31:30	OPST	Status: BIST (00), WAIT (01), ES16 (10), DEAD (11).
29:24	<i>reserved</i>	For future use by the RISC-V specification.
23:16	<i>custom</i>	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 29.4.3 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 29.4.2. 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]=0PST`:

- **00 - BIST** indicates that Built-In Self-Test "on-demand" (BIST) testing is being performed. If **0PST** 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.

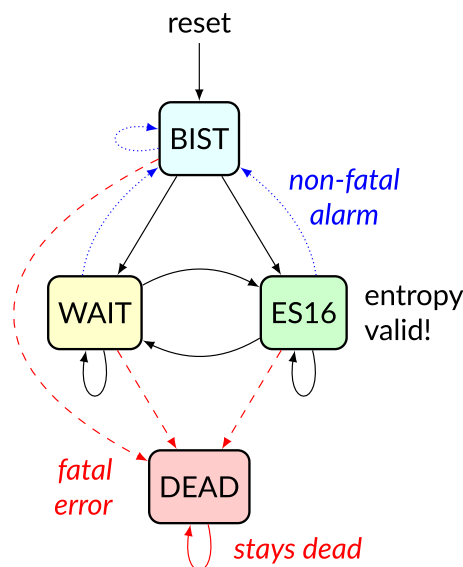


Figure 6. Entropy Source state transition diagram.

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.

29.4.2. 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 29.4.2.1](#): 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 29.4.2.2](#): 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 29.4.2.3](#): 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 29.7.6](#)

29.4.2.1. 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.

29.4.2.2. 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 29.4.2.1](#). 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.

29.4.2.3. 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.

29.4.3. 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 29.4.2.3) instead of invoking an SP 800-90B (Section 29.4.2.1) or PTG.2 (Section 29.4.2.2) *physical entropy source*. Emulated **seed** data generation is made with an appropriately seeded, secure software DRBG. See Section 29.7.3.5 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.

29.5. 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.

29.5.1. 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 chapter 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.

29.5.2. 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.

29.5.3. 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.

29.5.4. 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.

29.5.5. 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**.*

29.5.5.1. 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 <i>rd, rs1, rs2</i>

29.5.5.2. Zicond (Conditional Zero)

All instructions are included.

RV32	RV64	Mnemonic
✓	✓	czero.eqz <i>rd, rs1, rs2</i>
✓	✓	czero.nez <i>rd, rs1, rs2</i>

29.5.5.3. RVM (Multiply)

Multiplication is included; division and remaindering excluded.

RV32	RV64	Mnemonic
✓	✓	mul <i>rd, rs1, rs2</i>
✓	✓	mulh <i>rd, rs1, rs2</i>
✓	✓	mulhsu <i>rd, rs1, rs2</i>
✓	✓	mulhu <i>rd, rs1, rs2</i>
	✓	mulw <i>rd, rs1, rs2</i>

29.5.5.4. 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

29.5.5.5. Zcb Extension

These instructions are compressed versions of I and M instructions that are included in Zkt.

RV32	RV64	Mnemonic	Instruction
✓	✓	c.mul	Section 26.12.12
✓	✓	c.not	Section 26.12.11
✓	✓	c.zext.b	Section 26.12.6

29.5.5.6. 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 29.7.3.5](#).

RV32	RV64	Mnemonic	Instruction
✓		aes32dsi	Section 29.3.1
✓		aes32dsmi	Section 29.3.2
✓		aes32esi	Section 29.3.3
✓		aes32esmi	Section 29.3.4
	✓	aes64ds	Section 29.3.5
	✓	aes64dsm	Section 29.3.6
	✓	aes64es	Section 29.3.7
	✓	aes64esm	Section 29.3.8
	✓	aes64im	Section 29.3.9
	✓	aes64ks1i	Section 29.3.10
	✓	aes64ks2	Section 29.3.11
✓	✓	sha256sig0	Section 29.3.27
✓	✓	sha256sig1	Section 29.3.28
✓	✓	sha256sum0	Section 29.3.29
✓	✓	sha256sum1	Section 29.3.30
✓		sha512sig0h	Section 29.3.31
✓		sha512sig0l	Section 29.3.32
✓		sha512sig1h	Section 29.3.33
✓		sha512sig1l	Section 29.3.34
✓		sha512sum0r	Section 29.3.35
✓		sha512sum1r	Section 29.3.36
	✓	sha512sig0	Section 29.3.37
	✓	sha512sig1	Section 29.3.38
	✓	sha512sum0	Section 29.3.39
	✓	sha512sum1	Section 29.3.40
✓	✓	sm3p0	Section 29.3.41
✓	✓	sm3p1	Section 29.3.42
✓	✓	sm4ed	Section 29.3.43

RV32	RV64	Mnemonic	Instruction
✓	✓	sm4ks	Section 29.3.44

29.5.5.7. RVB (Bitmanip)

The [Section 29.2.1](#), [Section 29.2.2](#) and [Section 29.2.3](#) extensions are included in their entirety.

RV32	RV64	Mnemonic	Instruction
✓	✓	clmul	Section 29.3.14
✓	✓	clmulh	Section 29.3.15
✓	✓	xperm4	Section 29.3.48
✓	✓	xperm8	Section 29.3.47
✓	✓	ror	Section 29.3.23
✓	✓	rol	Section 29.3.21
✓	✓	rori	Section 29.3.24
	✓	rorw	Section 29.3.26
	✓	rolw	Section 29.3.22
	✓	roriw	Section 29.3.25
✓	✓	andn	Section 29.3.12
✓	✓	orn	Section 29.3.16
✓	✓	xnor	Section 29.3.46
✓	✓	pack	Section 29.3.17
✓	✓	packh	Section 29.3.18
	✓	packw	Section 29.3.19
✓	✓	brev8	Section 29.3.13
✓	✓	rev8	Section 29.3.20
✓		zip	Section 29.3.49
✓		unzip	Section 29.3.45

29.6. 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.

29.6.1. 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](#)).

29.6.2. 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.

29.6.3. 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.

29.6.4. 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 27](#)).



This section repeats much of the information in [Section 29.2.1](#), [Section 29.2.2](#) and [Section 29.2.3](#), 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.

29.6.4.1. Rotations

RV32, RV64:

```
ror    rd, rs1, rs2
rol    rd, rs1, rs2
rori   rd, rs1, imm
```

RV64 only:

```
rorw   rd, rs1, rs2
rolw   rd, rs1, rs2
roriw  rd, rs1, imm
```

See [Section 27.6](#) 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.

29.6.4.2. Bit & Byte Permutations

RV32, RV64:

```
brev8   rd, rs1
rev8    rd, rs1
```

See [Section 27.6](#) 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 [Section 27.6](#) 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.

29.6.4.3. Carry-less Multiply

RV32, RV64:

```
clmul    rd, rs1, rs2
clmulh   rd, rs1, rs2
```

See [Section 27.7](#) for details of these instructions. See [Section 29.5](#) 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.

29.6.4.4. Logic With Negate

RV32, RV64:

```
andn     rd, rs1, rs2
orn       rd, rs1, rs2
xnor     rd, rs1, rs2
```

See [Section 27.6](#) 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.

29.6.4.5. Packing

RV32, RV64:

```
pack    rd, rs1, rs2
packh   rd, rs1, rs2
```

RV64:

```
packw   rd, rs1, rs2
```

See [Section 27.6](#) 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.*

29.6.4.6. Crossbar Permutation Instructions

RV32, RV64:

```
xperm4  rd, rs1, rs2
xperm8  rd, rs1, rs2
```

See [Section 27.8](#) 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 ^[4]. Note that this technique is also suitable for masking based side-channel countermeasures.

29.7. 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.

29.7.1. 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 29.4](#) and [Section 29.7.6](#)), access control ([Section 29.4.3](#)), 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 29.7.4](#)), a conditioner ([Section 29.7.2.2](#)), 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-Shell-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 29.7.6](#) can be used 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 29.4.2](#). 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 29.7.5](#) 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

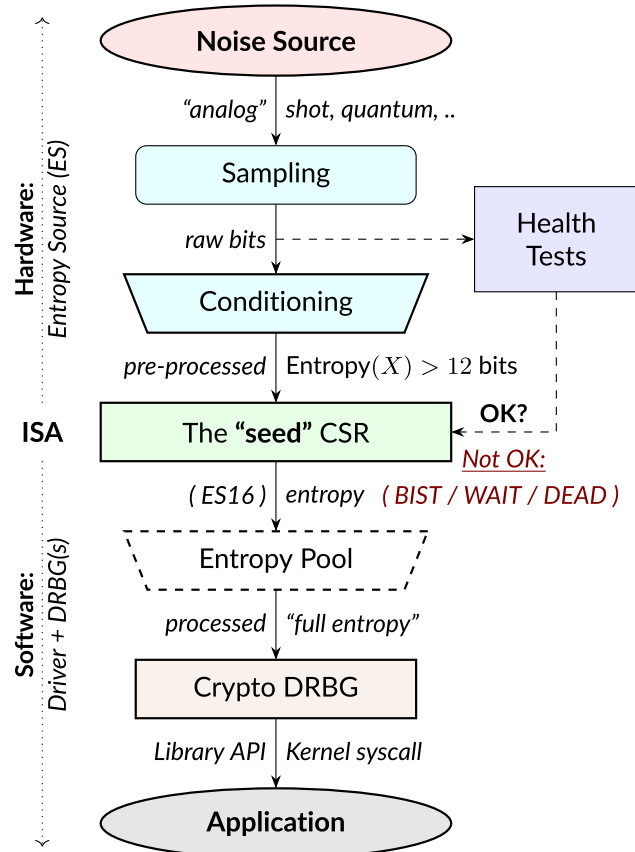
RISC-V processor design. Compared to older (FIPS 140-2) RNG and DRBG modules, an entropy source module may have a relatively small area (just a few thousand NAND2 gate equivalent). CMVP is introducing an "Entropy Source Validation Scope" which potentially allows 90B validations to be reused for different (FIPS 140-3) modules.

29.7.2. 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.

29.7.2.1. Entropy Source (ES)

Entropy sources are built by sampling and processing data from a noise source (Section 29.7.5.1). 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 29.4.2.1) for U.S. Federal FIPS 140-3 (NIST, 2019) evaluations and in BSI AIS-31 (Killmann & Schindler, 2001; Killmann & Schindler, 2011) (Section 29.4.2.2) 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.

29.7.2.2. 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 "debaised" 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 "debaised 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 29.7.5.5 for a more detailed discussion.

29.7.2.3. 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.

29.7.2.4. 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.

29.7.3. Specific Rationale and Considerations

29.7.3.1. The seed CSR

See Section 29.4.1.

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 **0PST = 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 29.7.5.6](#)), and the desire to have all of the bits "in the same place" on both RV32 and RV64 architectures for programming convenience.

29.7.3.2. NIST SP 800-90B

See [Section 29.4.2.1](#).

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.

29.7.3.3. BSI AIS-31

See [Section 29.4.2.2](#).

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 29.7.4](#)) 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 29.7.6](#)).
- 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.

29.7.3.4. Virtual Sources

[Section 29.4.2.3.](#)

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](#)).

29.7.3.5. Security Considerations for Direct Hardware Access

[Section 29.4.3.](#)

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 `msecfg.[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.

29.7.4. 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 its 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.

29.7.4.1. 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).

29.7.4.2. 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.

29.7.4.3. 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.

29.7.4.4. 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 (Evyushkin & 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.

29.7.5. 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).

29.7.5.1. 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.

29.7.5.2. 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).

29.7.5.3. 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 29.7.5.8 for a discussion of Quantum entropy sources.

29.7.5.4. 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 $W \leq 1024$ 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 29.7.4) 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.

29.7.5.5. Non-cryptographic Conditioners

As noted in Section 29.7.2.2, physical randomness sources generally require a post-processing step called *conditioning* to meet the desired quality requirements, which are outlined in Section 29.4.2.

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 29.4.2).

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.

29.7.5.6. 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.

29.7.5.7. 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.

29.7.5.8. 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.

29.7.5.9. 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.

29.7.6. 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.

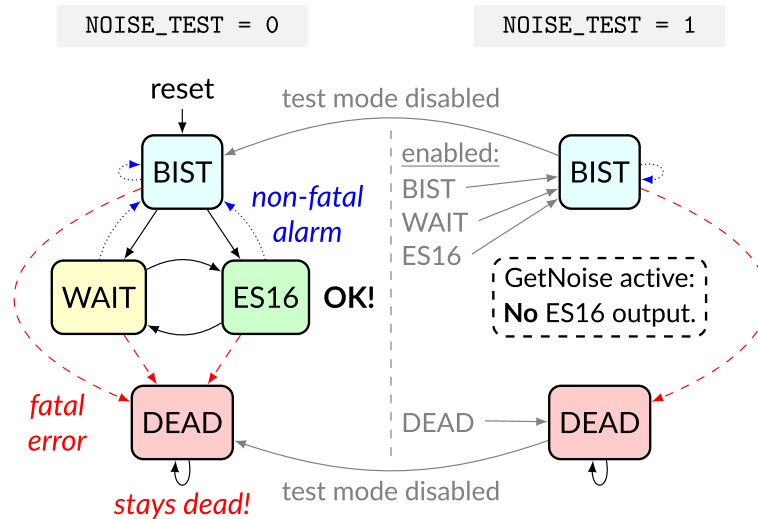


Figure 7. Entropy source can't be read in test mode.

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.

29.8. Supplementary Materials

While this chapter contains the specifications for the RISC-V cryptography extensions, numerous supplementary materials and example codes have also been developed. All of the supplementary materials related to the RISC-V Cryptography extension live in a GitHub Repository, located at github.com/riscv/riscv-crypto

- **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.

29.9. 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 sbbox_lookup : (bits(8), vector(256, bits(8))) -> bits(8)
function sbbox_lookup(x, table) = {
    table[255 - unsigned(x)]
}

/* Easy function to perform a forward AES SBox operation on 1 byte. */
val aes_sbbox_fwd : bits(8) -> bits(8)
function aes_sbbox_fwd(x) = sbbox_lookup(x, aes_sbbox_fwd_table)

/* Easy function to perform an inverse AES SBox operation on 1 byte. */
val aes_sbbox_inv : bits(8) -> bits(8)
function aes_sbbox_inv(x) = sbbox_lookup(x, aes_sbbox_inv_table)

/* AES SubWord function used in the key expansion
 * - Applies the forward sbbox to each byte in the input word.
 */
val aes_subword_fwd : bits(32) -> bits(32)
function aes_subword_fwd(x) = {
    aes_sbbox_fwd(x[31..24]) @
    aes_sbbox_fwd(x[23..16]) @
    aes_sbbox_fwd(x[15.. 8]) @
    aes_sbbox_fwd(x[ 7.. 0])
}

/* AES Inverse SubWord function.
 * - Applies the inverse sbbox to each byte in the input word.
 */
val aes_subword_inv : bits(32) -> bits(32)
function aes_subword_inv(x) = {
    aes_sbbox_inv(x[31..24]) @
    aes_sbbox_inv(x[23..16]) @
    aes_sbbox_inv(x[15.. 8]) @
    aes_sbbox_inv(x[ 7.. 0])
}

/* Easy function to perform an SM4 SBox operation on 1 byte. */
val sm4_sbbox : bits(8) -> bits(8)
function sm4_sbbox(x) = sbbox_lookup(x, sm4_sbbox_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 sbbox to each byte
 * in a 64-bit word.
 */
val aes_apply_fwd_sbbox_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 30. Cryptography Extensions: Vector Instructions, Version 1.0

This chapter describes the Vector Cryptography extensions to the RISC-V Instruction Set Architecture.

30.1. Introduction

This chapter 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. [Chapter 29](#) describes cryptographic instructions for smaller cores which do not implement the vector extension.

30.1.1. 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.

30.1.2. 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 30.6](#).

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](#).

30.1.3. 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.

30.1.4. 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 28.19](#).

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	Section 30.2.5	128	32	4
SHA256	zvknh[ab]	128	32	4
SHA512	zvknhb	256	64	4
GCM	Section 30.2.4	128	32	4
SM4	Section 30.2.7	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 30.1.5](#) for limitations on **vL** and **vstart** for vector crypto instructions.

30.1.5. 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 30.1.4](#)) 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 $EMUL = \text{ceil}(EGW / VLEN)$.



Scalar element group operands do not need to be aligned to LMUL for any implementation with $VLEN \geq EGW$.

In the case of the .vs instructions defined in this specification, **vs2** holds a 128-bit scalar element group. For implementations with $VLEN \geq 128$, **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

30.1.6. 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 30.1.4](#)) 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.

30.1.7. 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 **v1** 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	v1/2
SM3	vsm3*	32	v1/4

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.

30.2. Extensions Overview

The section introduces all of the extensions in the Vector Cryptography Instruction Set Extension Specification.

The **Zvknhb** and [Section 30.2.2](#) 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**.

Zvknhb implies **Zvknha**.

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 [Section 30.2.5](#), [Zvknh\[ab\]](#), [Zvkg](#), [Zvksed](#) and [Section 30.2.8](#) but *not* [Section 30.2.1](#), [Section 30.2.3](#), or [Section 30.2.2](#)) 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 [Section 30.2.1](#), [Section 30.2.3](#), and [Section 30.2.2](#)) 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.

30.2.1. Zvbb - Vector Basic Bit-manipulation

Vector basic bit-manipulation instructions.



This extension is a superset of the [Zvkb](#) extension.

Mnemonic	Instruction
vandn.[vv,vx]	Section 30.3.8
vbrev.v	Section 30.3.9
vbrev8.v	Section 30.3.10
vrev8.v	Section 30.3.18
vclz.v	Section 30.3.13
vctz.v	Section 30.3.15
vcpop.v	Section 30.3.14
vrol.[vv,vx]	Section 30.3.19
vror.[vv,vx,vi]	Section 30.3.20
vwsll.[vv,vx,vi]	Section 30.3.27

30.2.2. **zvb** - 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]	Section 30.3.11
vclmulh.[vv,vx]	Section 30.3.12

DRAFT

30.2.3. **zvk**b**** - 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]	Section 30.3.8
vbrev8.v	Section 30.3.10
vrev8.v	Section 30.3.18
vrol.[vv,vx]	Section 30.3.19
vror.[vv,vx,vi]	Section 30.3.20

DRAFT

30.2.4. **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 $\text{SEW}=32$ elements to be processed and therefore must be a multiple of $\text{EGS}=4$.

Likewise, vstart must be a multiple of $\text{EGS}=4$.

SEW	EGW	Mnemonic	Instruction
32	128	vghsh.vv	Section 30.3.16
32	128	vgmul.vv	Section 30.3.17

30.2.5. **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]	Section 30.3.3
32	128	vaesem.[vv,vs]	Section 30.3.4
32	128	vaesdf.[vv,vs]	Section 30.3.1
32	128	vaesdm.[vv,vs]	Section 30.3.2
32	128	vaeskf1.vi	Section 30.3.5
32	128	vaeskf2.vi	Section 30.3.6
32	128	vaesz.vs	Section 30.3.7

30.2.6. `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	Section 30.3.22
vsha2c[hl].vv	Section 30.3.21

30.2.7. `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	<code>vsm4k.vi</code>	Section 30.3.25
32	128	<code>vsm4r.[vv,vs]</code>	Section 30.3.26

DRAFT

30.2.8. 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 $v1/EGS$. $v1$ 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	Section 30.3.24
32	256	vsm3c.vi	Section 30.3.23

30.2.9. 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.

DRAFT

30.2.10. `zvkn` - NIST Algorithm Suite with carry-less multiply

This extension is shorthand for the following set of other extensions:

Included Extension	Description
<code>Zvkn</code>	Zvkn
<code>Zvbc</code>	Zvbc



This extension combines the NIST Algorithm Suite with the vector carry-less multiply extension to enable AES-GCM.

DRAFT

30.2.11. 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.

DRAFT

30.2.12. 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.

DRAFT

30.2.13. **zvks** - 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.

DRAFT

30.2.14. Zvks - 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.

DRAFT

30.2.15. 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.

30.2.15.1. All Zvbb instructions

- vandn.v[vx]
- vclz.v
- vcpop.v
- vctz.v
- vbrev.v
- vbrev8.v
- vrev8.v
- vro1.v[vx]
- vror.v[vxi]
- vwsll.[vv,vx,vi]



All Zvbb instructions are also covered by DIEL as they are a proper subset of Zvbb

30.2.15.2. All Zvbc instructions

- vclmul[h].v[vx]

30.2.15.3. add/sub

- v[r]sub.v[vx]
- vadd.v[ivx]
- vsub.v[vx]

- `vwadd[u].[vw][vx]`
- `vwsb[u].[vw][vx]`

30.2.15.4. add/sub with carry

- `vadc.v[ivx]m`
- `vmadc.v[ivx][m]`
- `vmsbc.v[vx]m`
- `vsbc.v[vx]m`

30.2.15.5. compare and set

- `vmseq.v[vxi]`
- `vmsgt[u].v[xi]`
- `vmsle[u].v[xi]`
- `vmslt[u].v[xi]`
- `vmsne.v[ivx]`

30.2.15.6. copy

- `vmv.s.x`
- `vmv.v.[ivxs]`
- `vmv[1248]r.v`

30.2.15.7. extend

- `vsext.vf[248]`
- `vzext.vf[248]`

30.2.15.8. 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]`

30.2.15.9. multiply

- `vmul[h].v[vx]`

- `vmulh[s]u.v[vx]`
- `vwmul.v[vx]`
- `vwmul[s]u.v[vx]`

30.2.15.10. multiply-add

- `vmacc.v[vx]`
- `vmadd.v[vx]`
- `vnmsac.v[vx]`
- `vnmsub.v[vx]`
- `vwmacc.v[vx]`
- `vwmacc[s]u.v[vx]`
- `vwmaccus.vx`

30.2.15.11. Integer Merge

- `vmerge.v[ivx]m`

30.2.15.12. 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`

30.2.15.13. shift

- `vnsr[al].w[ivx]`
- `vsl.v[ivx]`
- `vsr[al].v[ivx]`

30.2.15.14. 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]*

30.3. Instructions

30.3.1. vaesdf.[vv,vs]

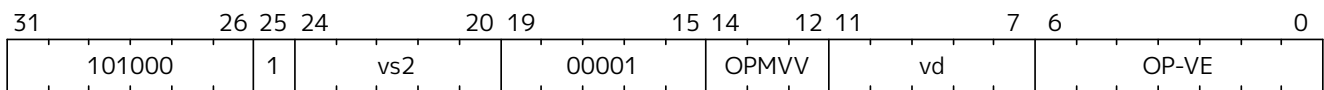
Synopsis

Vector AES final-round decryption

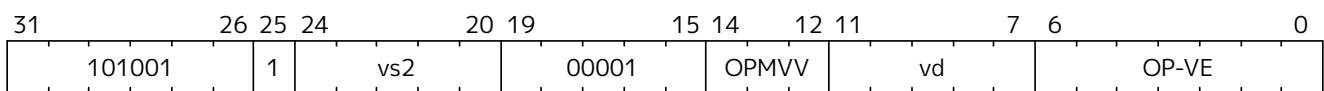
Mnemonic

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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

DRAFT

30.3.2. vaesdm.[vv,vs]

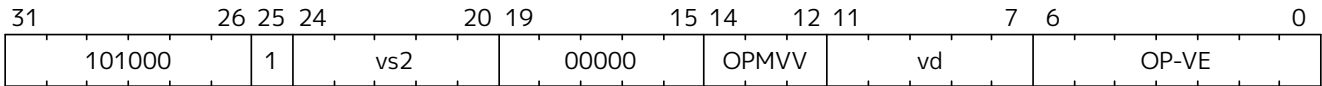
Synopsis

Vector AES middle-round decryption

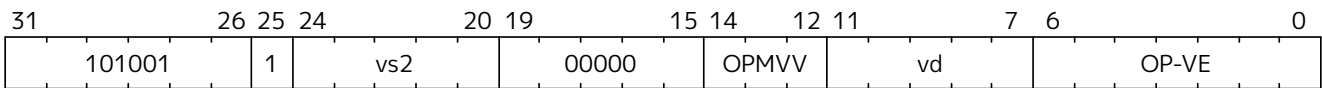
Mnemonic

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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

DRAFT

30.3.3. vaesef.[vv,vs]

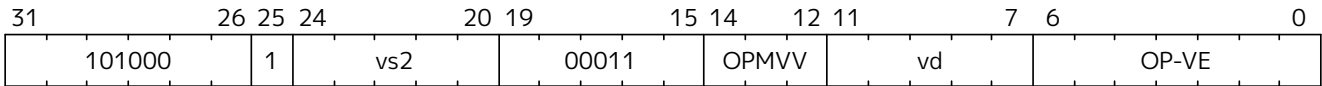
Synopsis

Vector AES final-round encryption

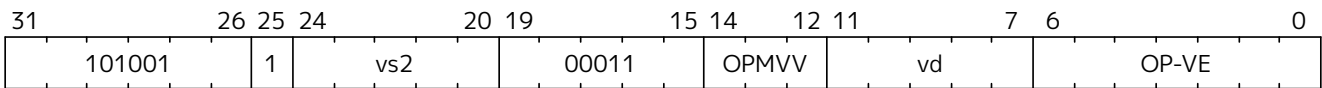
Mnemonic

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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

DRAFT

30.3.4. `vaesem.[vv,vs]`

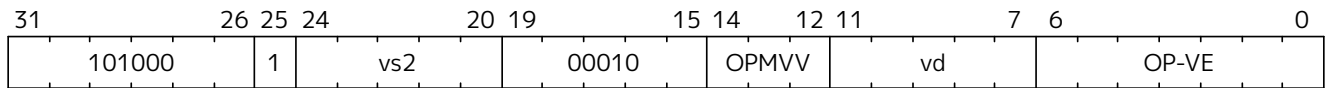
Synopsis

Vector AES middle-round encryption

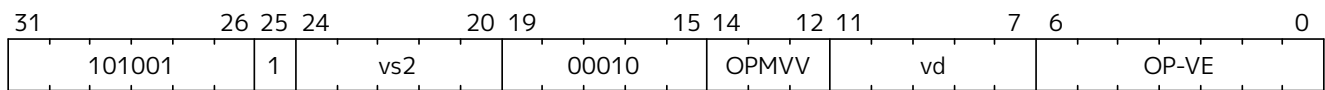
Mnemonic

`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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

DRAFT

30.3.5. vaeskfl.vi

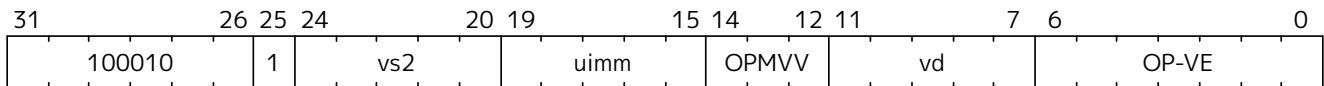
Synopsis

Vector AES-128 Forward KeySchedule generation

Mnemonic

vaeskfl.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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

30.3.6. vaeskf2.vi

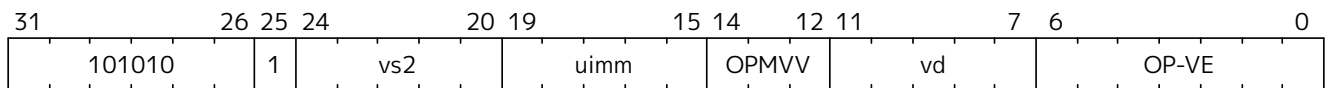
Synopsis

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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

30.3.7. vaesz.vs

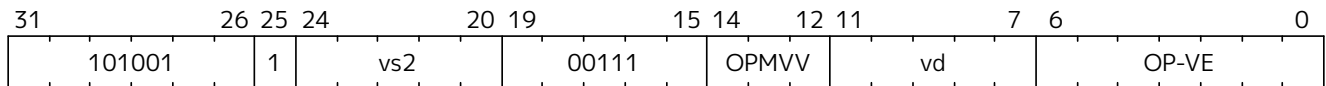
Synopsis

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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.5](#), [Section 30.2.11](#)

DRAFT

30.3.8. vandn.[vv,vx]

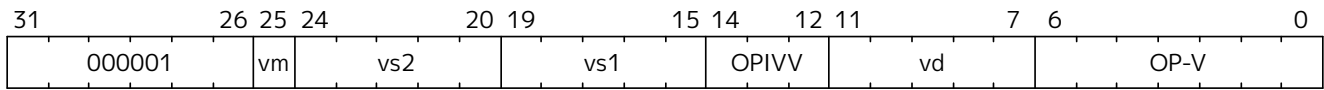
Synopsis

Bitwise And-Not

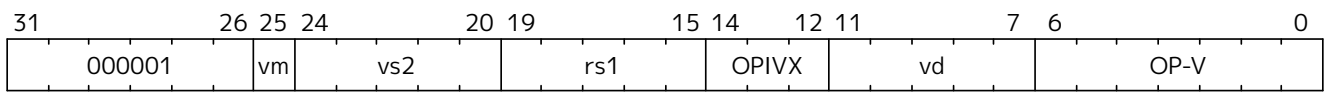
Mnemonic

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

[Section 30.2.1](#), [Section 30.2.3](#), [Section 30.2.9](#), [Section 30.2.10](#), [Zvkng](#), [Section 30.2.12 Zvksc](#), [Zvksg](#)

DRAFT

30.3.9. vbrev.v

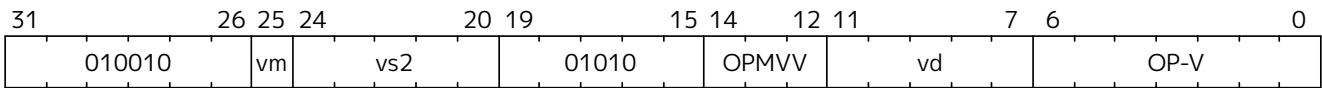
Synopsis

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

[Section 30.2.1](#)

30.3.10. vbrev8.v

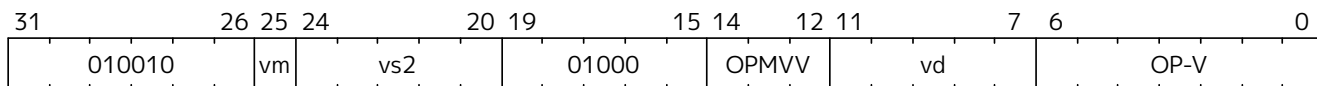
Synopsis

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

[Section 30.2.1](#), [Section 30.2.3](#), [Section 30.2.9](#), [Section 30.2.10](#), [Zvkng](#), [Section 30.2.12](#) [Zvksc](#), [Zvksg](#)

30.3.11. vclmul.[vv,vx]

Synopsis

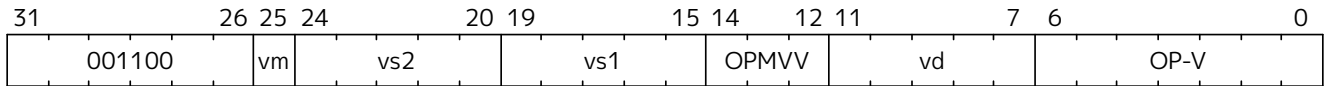
Vector Carry-less Multiply by vector or scalar - returning low half of product.

Mnemonic

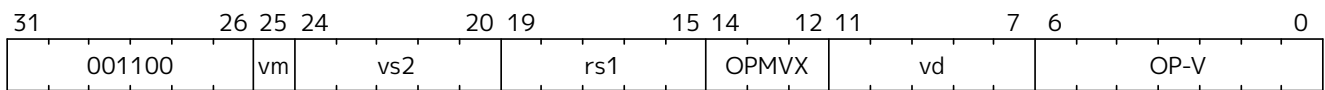
vclmul.vv vd, vs2, vs1, vm

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) = ctmul(op1,op2,SEW);
    set_velem(vd, i, product);
  }
  RETIRE_SUCCESS
}

function ctmul(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

[Section 30.2.2](#), [Section 30.2.10](#), [Section 30.2.13](#)

DRAFT

30.3.12. vclmulh.[vv,vx]

Synopsis

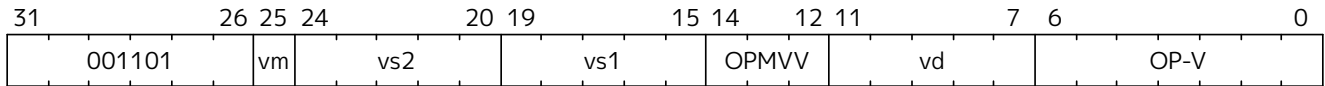
Vector Carry-less Multiply by vector or scalar - returning high half of product.

Mnemonic

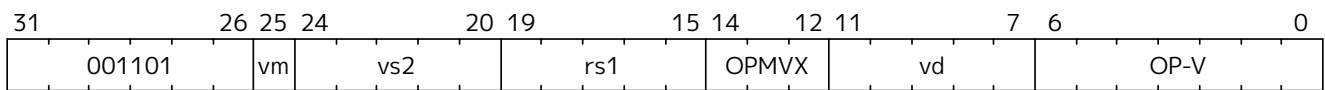
vclmulh.vv vd, vs2, vs1, vm

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

[Section 30.2.2](#), [Section 30.2.10](#), [Section 30.2.13](#)

DRAFT

30.3.13. vclz.v

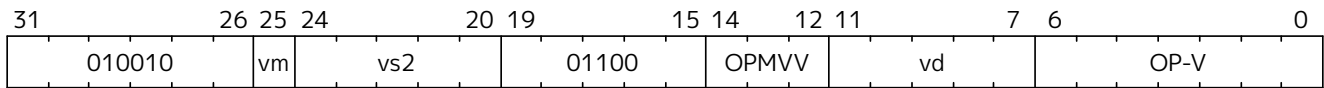
Synopsis

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

[Section 30.2.1](#)

30.3.14. vcpop.v

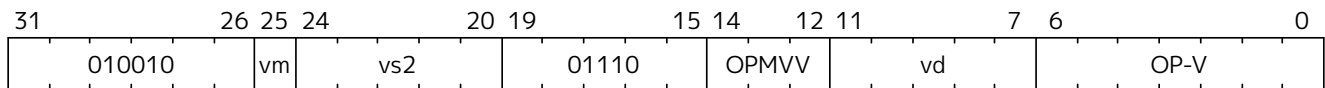
Synopsis

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 (VCP0P(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

[Section 30.2.1](#)

30.3.15. vctz.v

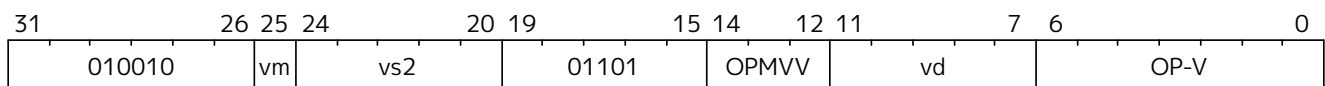
Synopsis

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

[Section 30.2.1](#)

DRAFT

30.3.16. vghsh.vv

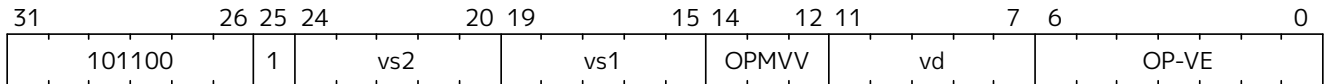
Synopsis

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 $\text{GF}[2]$. 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 $\text{GF}(2^{128})$) this sum by the Hash Subkey (H).

The multiplication over $\text{GF}(2^{128})$ 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 [Section 30.2.4](#)) orders the coefficients from left to right $x_0x_1x_2\dots x_{127}$ for a polynomial $x_0 + x_1u + x_2u^2 + \dots + x_{127}u^{127}$. 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 $\text{LMUL} > 1$.*

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

[Section 30.2.4](#), [Section 30.2.11](#), [Section 30.2.14](#)

30.3.17. vgmul.vv

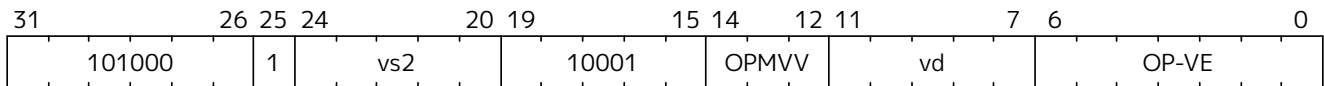
Synopsis

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 $\text{GF}[2]$. It produces the product over $\text{GF}(2^{128})$ of the two 128-bit inputs.

The multiplication over $\text{GF}(2^{128})$ 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 [Section 30.2.4](#)) orders the coefficients from left to right $x_0x_1x_2...x_{127}$ for a polynomial $x_0 + x_1u + x_2u^2 + ... + x_{127}u^{127}$. 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 **vgsh.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 **vgsh.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

[Section 30.2.4](#), [Section 30.2.11](#), [Section 30.2.14](#)

30.3.18. vrev8.v

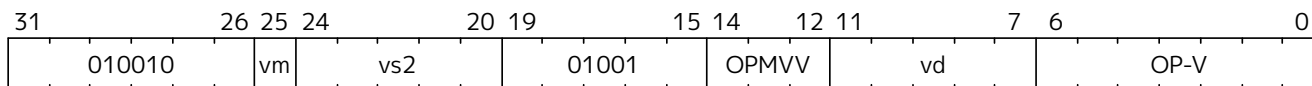
Synopsis

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

[Section 30.2.1](#), [Section 30.2.3](#), [Section 30.2.9](#), [Section 30.2.10](#), [Zvkng](#), [Section 30.2.12](#) [Zvksc](#), [Zvksg](#)

30.3.19. vrol.[vv,vx]

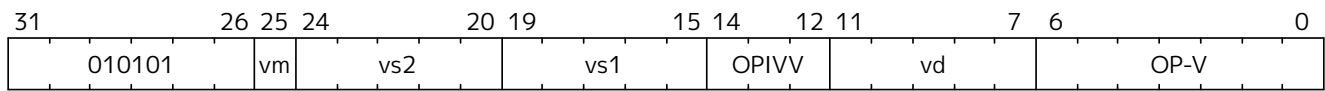
Synopsis

Vector rotate left by vector/scalar.

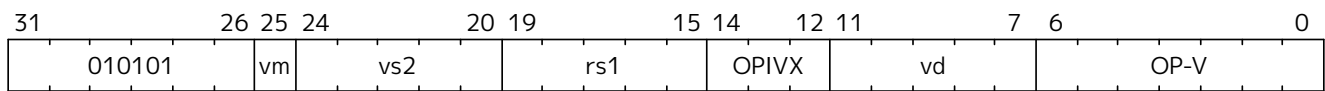
Mnemonic

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 $\log_2(\text{SEW})$ 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 (VR0L_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

[Section 30.2.1](#), [Section 30.2.3](#), [Section 30.2.9](#), [Section 30.2.10](#), [Zvkng](#), [Section 30.2.12 Zvksc](#), [Zvksg](#)

DRAFT

30.3.20. vror.[vv,vx,vi]

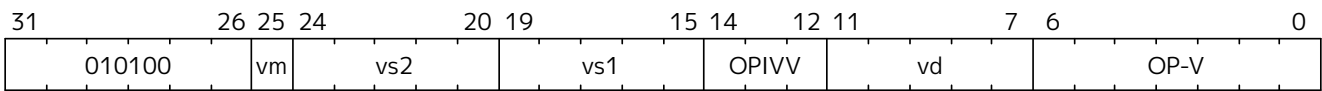
Synopsis

Vector rotate right by vector/scalar/immediate.

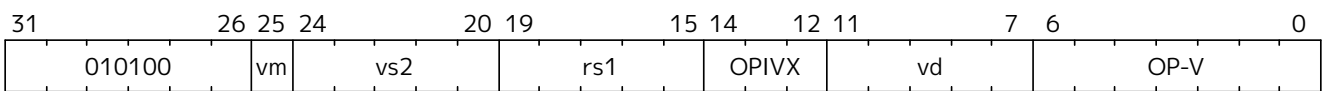
Mnemonic

vror.vv vd, vs2, vs1, vm
vror.vx vd, vs2, rs1, vm
vror.vi vd, vs2, uimm, vm

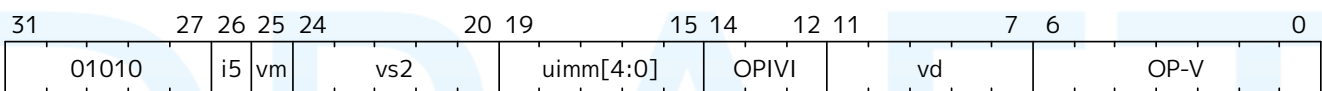
Encoding (Vector-Vector)



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

[Section 30.2.1](#), [Section 30.2.3](#), [Section 30.2.9](#), [Section 30.2.10](#), [Zvkng](#), [Section 30.2.12 Zvksc](#), [Zvksg](#)

30.3.21. vsha2c[hl].vv

Synopsis

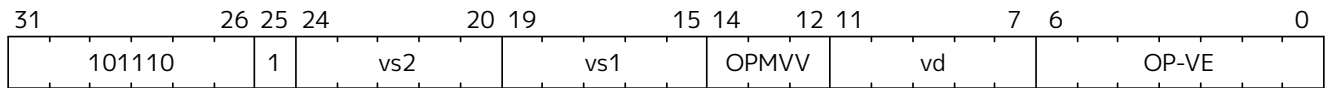
Vector SHA-2 two rounds of compression.

Mnemonic

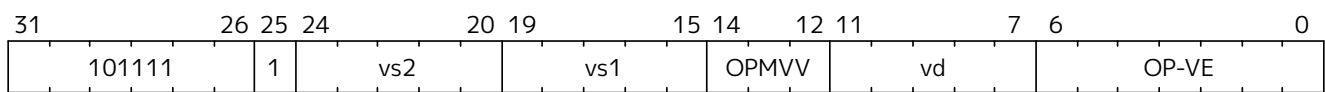
vsha2ch.vv vd, vs2, vs1

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 [Section 30.2.6](#)) 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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.11](#), [zvknh\[ab\]](#)

30.3.22. vsha2ms.vv

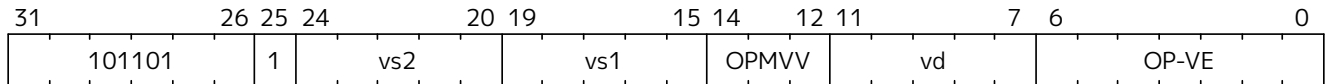
Synopsis

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 relative age with lower indices indicating older words.



Note to software developers

The {W₁₁, W₁₀, W₉, W₄} element group can easily be formed by using a vector **vmerge** instruction with the appropriate mask (for example with **v1=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

[Section 30.2.9](#), [Section 30.2.10](#), [Section 30.2.11](#), [zvknh\[ab\]](#)

DRAFT

30.3.23. vsm3c.vi

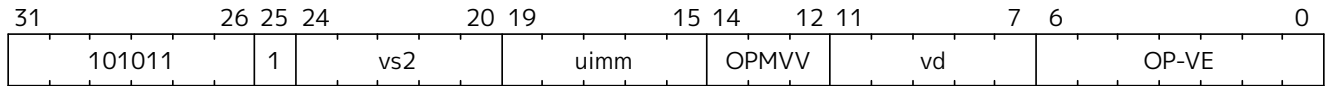
Synopsis

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

[Section 30.2.12](#), [Section 30.2.13](#), [Section 30.2.14](#), [Section 30.2.8](#)

30.3.24. vsm3me.vv

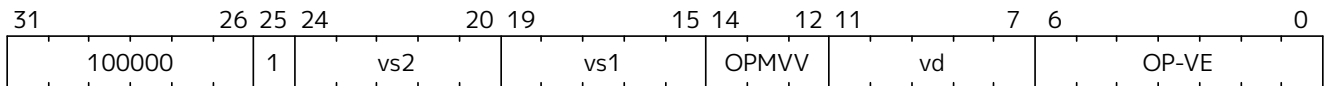
Synopsis

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

[Section 30.2.12](#), [Section 30.2.13](#), [Section 30.2.14](#), [Section 30.2.8](#)

DRAFT

30.3.25. vsm4k.vi

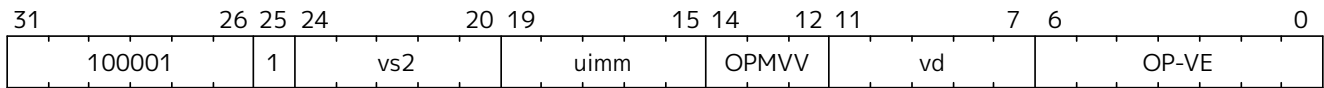
Synopsis

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:

$$\begin{aligned}
 B[n] &= n + 2n + 4n; \\
 &= 8n + \sim n + 1;
 \end{aligned}$$

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

[Section 30.2.12](#), [Section 30.2.13](#), [Section 30.2.7](#), [Section 30.2.14](#)

DRAFT

30.3.26. vsm4r.[vv,vs]

Synopsis

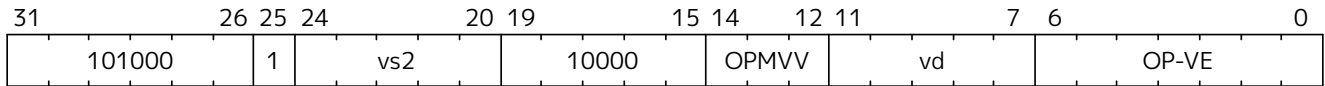
Vector SM4 Rounds

Mnemonic

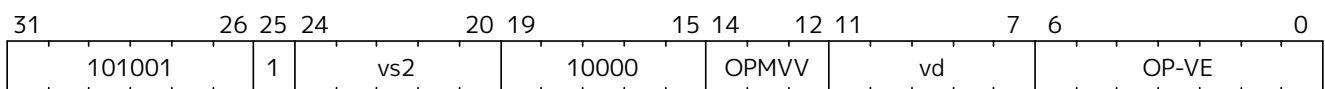
vsm4r.vv vd, vs2

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

[Section 30.2.12](#), [Section 30.2.13](#), [Section 30.2.7](#), [Section 30.2.14](#)

DRAFT

30.3.27. vwsll.[vv,vx,vi]

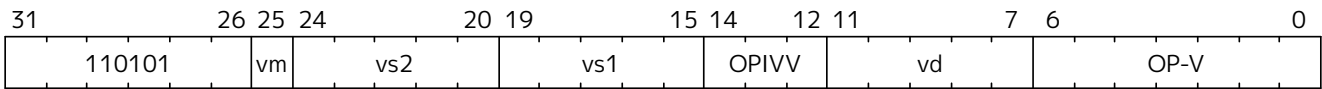
Synopsis

Vector widening shift left logical by vector/scalar/immediate.

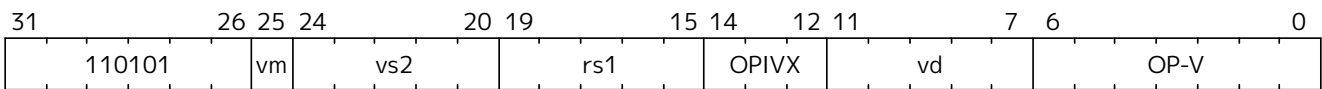
Mnemonic

vwsll.vv vd, vs2, vs1, vm
vwsll.vx vd, vs2, rs1, vm
vwsll.vi vd, vs2, uimm, vm

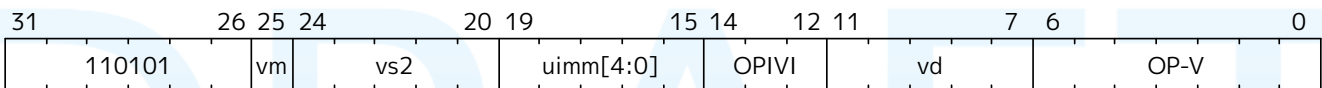
Encoding (Vector-Vector)



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
[Section 30.2.1](#)

30.4. 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

30.5. 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
000001	V	X		vandn	000001	V		vredand	000001	V	
000010	V	X		vsub	000010	V		vredor	000010	V	F
000011		X	I	vrsub	000011	V		vredxor	000011	V	
000100	V	X		vminu	000100	V		vredminu	000100	V	F
000101	V	X		vmin	000101	V		vredmin	000101	V	
000110	V	X		vmaxu	000110	V		vredmaxu	000110	V	F
000111	V	X		vmax	000111	V		vredmax	000111	V	
001000					001000	V	X	vaaddu	001000	V	F
001001	V	X	I	vand	001001	V	X	vaadd	001001	V	F
001010	V	X	I	vor	001010	V	X	vasubu	001010	V	F
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
001110	V			vrgatherei16							
001111		X	I	vslidedown	001111		X	vslide1down	001111		F

funct6				funct6				funct6			
010000	V	X	I	vadc	010000	V		VWXUNARYO	010000	V	
					010000		X	VRXUNARYO	010000		F

funct6				funct6				funct6			
010001	V	X	I	vmadc	010001			010001			
010010	V	X		vsbc	010010	V		010010	V		VFUNARYO
010011	V	X		vmsbc	010011			010011	V		VFUNARY1
010100	V	X		vrór	010100	V		010100			
010101	V	X		vrol	010101			010101			
01010x			I	vrór							
010110					010110			010110			
010111	V	X	I	vmerge/vmv	010111	V		010111		F	vfmerge/vfmv
011000	V	X	I	vmseq	011000	V		011000	V	F	vmfeq
011001	V	X	I	vmsne	011001	V		011001	V	F	vmfle
011010	V	X		vmsltu	011010	V		011010			
011011	V	X		vmslt	011011	V		011011	V	F	vmflt
011100	V	X	I	vmsleu	011100	V		011100	V	F	vmfne
011101	V	X	I	vmsle	011101	V		011101		F	vmfgt
011110		X	I	msgtu	011110	V		011110			
011111		X	I	msgt	011111	V		011111		F	vmfge

funct6				funct6				funct6			
100000	V	X	I	vsaddu	100000	V	X	100000	V	F	vfdiv
100001	V	X	I	vsadd	100001	V	X	100001		F	vfrdiv
100010	V	X		vssubu	100010	V	X	100010			
100011	V	X		vssub	100011	V	X	100011			
100100					100100	V	X	100100	V	F	vfmul
100101	V	X	I	vsll	100101	V	X	100101			
100110					100110	V	X	100110			
100111	V	X		vsmul	100111	V	X	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	101001	V	F	vfnmadd
101010	V	X	I	vssrl	101010			101010	V	F	vfmsub
101011	V	X	I	vssra	101011	V	X	101011	V	F	vfnmsub
101100	V	X	I	vnsrl	101100			101100	V	F	vfmacc
101101	V	X	I	vnsra	101101	V	X	101101	V	F	vfnmacc
101110	V	X	I	vnclipu	101110			101110	V	F	vfmsac
101111	V	X	I	vnclip	101111	V	X	101111	V	F	vfnmsac

funct6				funct6				funct6			
110000	V			vwredsumu	110000	V	X	110000	V	F	vwadd
110001	V			vwredsum	110001	V	X	110001	V		vwredusum
110010					110010	V	X	110010	V	F	vwsub

funct6				funct6				funct6			
110011				110011	V	X	vwsb	110011	V		vfwredosum
110100				110100	V	X	vwaddu.w	110100	V	F	vfwadd.w
110101	V	X	I	110101	V	X	vwadd.w	110101			
110110				110110	V	X	vwsbu.w	110110	V	F	vfwsub.w
110111				110111	V	X	vwsb.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	vfwmac
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

DRAFT

Table 71. VXUNARYO 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

30.6. 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])
}

```

DRAFT

Chapter 31. "Zvknhk" Extension for Vector Keccak

This chapter describes the **Zvknhk** Vector Cryptography extension, which adds vector instructions to compute the Keccak sponge function using in NIST SHA3 and numerous Post Quantum Cryptography (PQC) algorithms.

This document is in *Development* state. Assume that everything can change. This document has not been publicly reviewed nor approved yet. For more information, see [here](#).

31.1. Extension Overview

This section introduces **Zvknhk**.

Zvknhk depends on **Zve64x**.

As the instructions defined in this specification might be used to implement cryptographic primitives, they must be implemented with data-independent execution latencies as defined in [link:https://github.com/riscv/riscv-isa-manual/blob/main/src/scalar-crypto.adoc#data-independent-execution-latency-subset-zkt](https://github.com/riscv/riscv-isa-manual/blob/main/src/scalar-crypto.adoc#data-independent-execution-latency-subset-zkt).

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.

Zvknhk defines instructions to enable the efficient implementation of the Keccak sponge function.

The instructions defined in this specification share the following constraints:

- element group size (EGS) is 32
- data independent execution timing
- **vL/vstart** must be multiples of EGS=32 (else behavior is *reserved*)

All of these instructions work on 2048-bit element groups comprised of 32 64-bit elements, in element group parlance **EGS=32**, **EGW=2048** and the instructions are only defined for **SEW=64**.

Note

The value of EGS makes the current state of this specification unsuitable for implementation with **VLEN** < 256 (including **VLEN=128** which is the current minimum mandate in RVA profiles). This is a known limitation which the PQC task group intends to address before finalizing the specification.

Note

The DIEL mandate is independent from the support of **Zvkt**: **Zvknhk** defines instructions used primarily for cryptography and there are of little use if their execution latency can leak information about input data.

The number of element groups to be processed is **vL/EGS**. **vL** must be set to the number of **SEW=64** elements to be processed and therefore must be a multiple of **EGS=32**. Otherwise the behavior is *reserved*. Likewise, **vstart** must be a multiple of **EGS=32**. Otherwise the behavior is *reserved*.

Zvknhk defines the following instructions:

SEW	EGW	Mnemonic	Instruction
64	2048	vkeccak.vi	Section 31.2

31.2. vkeccak.vi

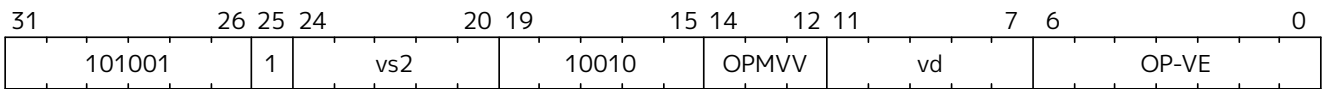
Synopsis

Vector-Immediate Multi-Round Keccak

Mnemonic

vkeccak.vi vd, imm5

Encoding (Vector-Immediate)



Reserved Encodings

- SEW is any value other than 64

Arguments

Register	Direction	EGW	EGS	SEW	Definition
vd	input	2048	32	64	Partial hash
imm5	input	-	-	-	Round Number (Immediate)
vd	output	2048	32	64	Partial hash

Description

The instructions performs the number of Keccak round(s) indicated by the immediate value `imm5` plus one (1-32 round(s)) on the state contained in the `vd` vector register group. The final result is written back into the same `vd` register group.

Note

out of the 32 64-bit elements in each element groups only the lower 25 are used. The upper 7 elements follows the tail policy.

Note

we could leave the behavior on the upper 7 elements of each element groups implementation defined.

Operation

```

function clause execute (VKECCAKVI(vs2, 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)
    }
}

```

```

// TODO need to describe properly the iteration over element group(s)
foreach (i from eg_start to eg_len-1) {
    const uint32_t roundCnt = zimm5;,
{
    // For VKECCAK
    // - vd contains the input state,
    // - vs2 contains the round index,
    // - vd does receive the output state.
    //
    // note that in the 32 elements of each EGU64x32 only the first 25
    // elements are actually used (the Keccak state is 1600-bit wide)
    EGU64x32_t keccak_state = P.VU.elc_group<EGU64x32_t>(vd_num, idx_eg);

    // unpacking state
    uint64_t A_0_0 = keccak_state[0 + 5 * 0];
    uint64_t A_0_1 = keccak_state[0 + 5 * 1];
    uint64_t A_0_2 = keccak_state[0 + 5 * 2];
    uint64_t A_0_3 = keccak_state[0 + 5 * 3];
    uint64_t A_0_4 = keccak_state[0 + 5 * 4];
    uint64_t A_1_0 = keccak_state[1 + 5 * 0];
    uint64_t A_1_1 = keccak_state[1 + 5 * 1];
    uint64_t A_1_2 = keccak_state[1 + 5 * 2];
    uint64_t A_1_3 = keccak_state[1 + 5 * 3];
    uint64_t A_1_4 = keccak_state[1 + 5 * 4];
    uint64_t A_2_0 = keccak_state[2 + 5 * 0];
    uint64_t A_2_1 = keccak_state[2 + 5 * 1];
    uint64_t A_2_2 = keccak_state[2 + 5 * 2];
    uint64_t A_2_3 = keccak_state[2 + 5 * 3];
    uint64_t A_2_4 = keccak_state[2 + 5 * 4];
    uint64_t A_3_0 = keccak_state[3 + 5 * 0];
    uint64_t A_3_1 = keccak_state[3 + 5 * 1];
    uint64_t A_3_2 = keccak_state[3 + 5 * 2];
    uint64_t A_3_3 = keccak_state[3 + 5 * 3];
    uint64_t A_3_4 = keccak_state[3 + 5 * 4];
    uint64_t A_4_0 = keccak_state[4 + 5 * 0];
    uint64_t A_4_1 = keccak_state[4 + 5 * 1];
    uint64_t A_4_2 = keccak_state[4 + 5 * 2];
    uint64_t A_4_3 = keccak_state[4 + 5 * 3];
    uint64_t A_4_4 = keccak_state[4 + 5 * 4];
    // executed the number of requested keccak rounds
    for (std::size_t ridx = 0; ridx < roundCnt; ++ridx) {
        uint64_t C_0= A_0_0 ^ A_0_1 ^ A_0_2 ^ A_0_3 ^ A_0_4;
        uint64_t C_1= A_1_0 ^ A_1_1 ^ A_1_2 ^ A_1_3 ^ A_1_4;
        uint64_t C_2= A_2_0 ^ A_2_1 ^ A_2_2 ^ A_2_3 ^ A_2_4;
        uint64_t C_3= A_3_0 ^ A_3_1 ^ A_3_2 ^ A_3_3 ^ A_3_4;
        uint64_t C_4= A_4_0 ^ A_4_1 ^ A_4_2 ^ A_4_3 ^ A_4_4;
        uint64_t D_0 = C_4 ^ KECCAK_ROL(C_1,1);
        A_0_0 ^= D_0;
        A_0_1 ^= D_0;

```

```

A_0_2 ^= D_0;
A_0_3 ^= D_0;
A_0_4 ^= D_0;
uint64_t D_1 = C_0 ^ KECCAK_ROL(C_2,1);
A_1_0 ^= D_1;
A_1_1 ^= D_1;
A_1_2 ^= D_1;
A_1_3 ^= D_1;
A_1_4 ^= D_1;
uint64_t D_2 = C_1 ^ KECCAK_ROL(C_3,1);
A_2_0 ^= D_2;
A_2_1 ^= D_2;
A_2_2 ^= D_2;
A_2_3 ^= D_2;
A_2_4 ^= D_2;
uint64_t D_3 = C_2 ^ KECCAK_ROL(C_4,1);
A_3_0 ^= D_3;
A_3_1 ^= D_3;
A_3_2 ^= D_3;
A_3_3 ^= D_3;
A_3_4 ^= D_3;
uint64_t D_4 = C_3 ^ KECCAK_ROL(C_0,1);
A_4_0 ^= D_4;
A_4_1 ^= D_4;
A_4_2 ^= D_4;
A_4_3 ^= D_4;
A_4_4 ^= D_4;
uint64_t T_0 = A_1_0;
uint64_t T_1 = A_0_2;
A_0_2 = KECCAK_ROL(T_0, 1);
uint64_t T_2 = A_2_1;
A_2_1 = KECCAK_ROL(T_1, 3);
uint64_t T_3 = A_1_2;
A_1_2 = KECCAK_ROL(T_2, 6);
uint64_t T_4 = A_2_3;
A_2_3 = KECCAK_ROL(T_3, 10);
uint64_t T_5 = A_3_3;
A_3_3 = KECCAK_ROL(T_4, 15);
uint64_t T_6 = A_3_0;
A_3_0 = KECCAK_ROL(T_5, 21);
uint64_t T_7 = A_0_1;
A_0_1 = KECCAK_ROL(T_6, 28);
uint64_t T_8 = A_1_3;
A_1_3 = KECCAK_ROL(T_7, 36);
uint64_t T_9 = A_3_1;
A_3_1 = KECCAK_ROL(T_8, 45);
uint64_t T_10 = A_1_4;
A_1_4 = KECCAK_ROL(T_9, 55);
uint64_t T_11 = A_4_4;
A_4_4 = KECCAK_ROL(T_10, 2);

```



```

uint64_t T_12 = A_4_0;
A_4_0 = KECCAK_ROL(T_11, 14);
uint64_t T_13 = A_0_3;
A_0_3 = KECCAK_ROL(T_12, 27);
uint64_t T_14 = A_3_4;
A_3_4 = KECCAK_ROL(T_13, 41);
uint64_t T_15 = A_4_3;
A_4_3 = KECCAK_ROL(T_14, 56);
uint64_t T_16 = A_3_2;
A_3_2 = KECCAK_ROL(T_15, 8);
uint64_t T_17 = A_2_2;
A_2_2 = KECCAK_ROL(T_16, 25);
uint64_t T_18 = A_2_0;
A_2_0 = KECCAK_ROL(T_17, 43);
uint64_t T_19 = A_0_4;
A_0_4 = KECCAK_ROL(T_18, 62);
uint64_t T_20 = A_4_2;
A_4_2 = KECCAK_ROL(T_19, 18);
uint64_t T_21 = A_2_4;
A_2_4 = KECCAK_ROL(T_20, 39);
uint64_t T_22 = A_4_1;
A_4_1 = KECCAK_ROL(T_21, 61);
uint64_t T_23 = A_1_1;
A_1_1 = KECCAK_ROL(T_22, 20);
uint64_t T_24 = A_1_0;
A_1_0 = KECCAK_ROL(T_23, 44);
uint64_t C_0_0 = A_0_0;
uint64_t C_0_1 = A_1_0;
uint64_t C_0_2 = A_2_0;
uint64_t C_0_3 = A_3_0;
uint64_t C_0_4 = A_4_0;
A_0_0 = C_0_0 ^ (~C_0_1 & C_0_2);
A_1_0 = C_0_1 ^ (~C_0_2 & C_0_3);
A_2_0 = C_0_2 ^ (~C_0_3 & C_0_4);
A_3_0 = C_0_3 ^ (~C_0_4 & C_0_0);
A_4_0 = C_0_4 ^ (~C_0_0 & C_0_1);
uint64_t C_1_0 = A_0_1;
uint64_t C_1_1 = A_1_1;
uint64_t C_1_2 = A_2_1;
uint64_t C_1_3 = A_3_1;
uint64_t C_1_4 = A_4_1;
A_0_1 = C_1_0 ^ (~C_1_1 & C_1_2);
A_1_1 = C_1_1 ^ (~C_1_2 & C_1_3);
A_2_1 = C_1_2 ^ (~C_1_3 & C_1_4);
A_3_1 = C_1_3 ^ (~C_1_4 & C_1_0);
A_4_1 = C_1_4 ^ (~C_1_0 & C_1_1);
uint64_t C_2_0 = A_0_2;
uint64_t C_2_1 = A_1_2;
uint64_t C_2_2 = A_2_2;
uint64_t C_2_3 = A_3_2;

```

```

uint64_t C_2_4 = A_4_2;
A_0_2 = C_2_0 ^ (~C_2_1 & C_2_2);
A_1_2 = C_2_1 ^ (~C_2_2 & C_2_3);
A_2_2 = C_2_2 ^ (~C_2_3 & C_2_4);
A_3_2 = C_2_3 ^ (~C_2_4 & C_2_0);
A_4_2 = C_2_4 ^ (~C_2_0 & C_2_1);
uint64_t C_3_0 = A_0_3;
uint64_t C_3_1 = A_1_3;
uint64_t C_3_2 = A_2_3;
uint64_t C_3_3 = A_3_3;
uint64_t C_3_4 = A_4_3;
A_0_3 = C_3_0 ^ (~C_3_1 & C_3_2);
A_1_3 = C_3_1 ^ (~C_3_2 & C_3_3);
A_2_3 = C_3_2 ^ (~C_3_3 & C_3_4);
A_3_3 = C_3_3 ^ (~C_3_4 & C_3_0);
A_4_3 = C_3_4 ^ (~C_3_0 & C_3_1);
uint64_t C_4_0 = A_0_4;
uint64_t C_4_1 = A_1_4;
uint64_t C_4_2 = A_2_4;
uint64_t C_4_3 = A_3_4;
uint64_t C_4_4 = A_4_4;
A_0_4 = C_4_0 ^ (~C_4_1 & C_4_2);
A_1_4 = C_4_1 ^ (~C_4_2 & C_4_3);
A_2_4 = C_4_2 ^ (~C_4_3 & C_4_4);
A_3_4 = C_4_3 ^ (~C_4_4 & C_4_0);
A_4_4 = C_4_4 ^ (~C_4_0 & C_4_1);
/* */ // XL(0,0,RC[i]);
A_0_0 ^= KECCAK_RC[ridx];
};
// epilog: populating back keccak state
keccak_state[0 + 5 * 0] = A_0_0;
keccak_state[0 + 5 * 1] = A_0_1;
keccak_state[0 + 5 * 2] = A_0_2;
keccak_state[0 + 5 * 3] = A_0_3;
keccak_state[0 + 5 * 4] = A_0_4;
keccak_state[1 + 5 * 0] = A_1_0;
keccak_state[1 + 5 * 1] = A_1_1;
keccak_state[1 + 5 * 2] = A_1_2;
keccak_state[1 + 5 * 3] = A_1_3;
keccak_state[1 + 5 * 4] = A_1_4;
keccak_state[2 + 5 * 0] = A_2_0;
keccak_state[2 + 5 * 1] = A_2_1;
keccak_state[2 + 5 * 2] = A_2_2;
keccak_state[2 + 5 * 3] = A_2_3;
keccak_state[2 + 5 * 4] = A_2_4;
keccak_state[3 + 5 * 0] = A_3_0;
keccak_state[3 + 5 * 1] = A_3_1;
keccak_state[3 + 5 * 2] = A_3_2;
keccak_state[3 + 5 * 3] = A_3_3;
keccak_state[3 + 5 * 4] = A_3_4;

```

```

    keccak_state[4 + 5 * 0] = A_4_0;
    keccak_state[4 + 5 * 1] = A_4_1;
    keccak_state[4 + 5 * 2] = A_4_2;
    keccak_state[4 + 5 * 3] = A_4_3;
    keccak_state[4 + 5 * 4] = A_4_4;

    // Update the destination register.
    EGU64x32_t &vd = P.VU.el_t_group<EGU64x32_t>(vd_num, idx_eg, true);
    EGU64x32_COPY(vd, keccak_state);
}
);
    RETIRE_SUCCESS
}
}

```

Included in

[Chapter 31](#)

31.3. Encodings

Appendix A: Additional Vector Cryptographic Instructions

OP-VE (0x77) Vector Crypto instructions, including **Zvknkk**, except **Zvbb** and **Zvbc**. The new/modified encodings are 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				
100000					100000	V	vsm3me			100000				
100001					100001	V	vsm4k.vi			100001				
100010					100010	V	vaesfk1.vi			100010				
100011					100011	V	vghsh.vs			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	vaesfk2.vi			101010				
101011					101011	V	vsm3c.vi			101011				

funct6					funct6			funct6			
101100					101100	V	vghsh	101100			
101101					101101	V	vsha2ms	101101			
101110					101110	V	vsha2ch	101110			
101111					101111	V	vsha2cl	101111			

DRAFT

Table 72. VAES.vv and VAES.vs encoding space

vs1	
00000	vaesdm
00001	vaesdf
00010	vaesem
00011	vaesef
00111	vaesz
10000	vsm4r
10001	vgmul
10010	vkeccak

DRAFT

Chapter 32. 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., $rd \neq x0$. 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 32.1](#) and for the Unprivileged ISA for Zicfiss extension is specified in [Section 32.2](#). The Privileged ISA for these extensions is specified in the Privileged ISA specification.

32.1. 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 32.1.2](#)) 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.

32.1.1. 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.*

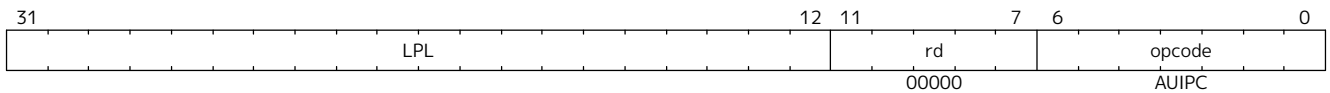
DRAFT

32.1.2. 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

<<<

32.2. 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.

32.2.1. Zicfiss Instructions Summary

The Zicfiss extension introduces the following instructions:

- Push to the shadow stack (See [Section 32.2.4](#))
 - **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 32.2.5](#))
 - **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 32.2.6](#))
 - **SSRDP** - encoded using **MOP.R.28**
- Perform an atomic swap from a shadow stack location (See [Section 32.2.7](#))
 - **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.

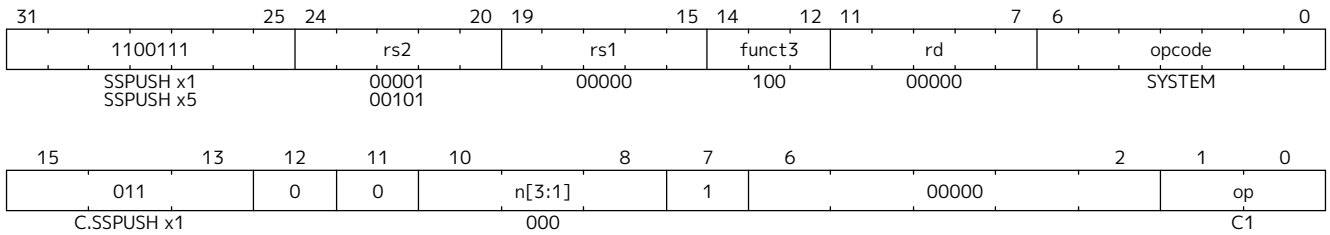
32.2.2. Shadow Stack Pointer (**ssp**)

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.

32.2.3. Zicfiss Instructions

32.2.4. 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:

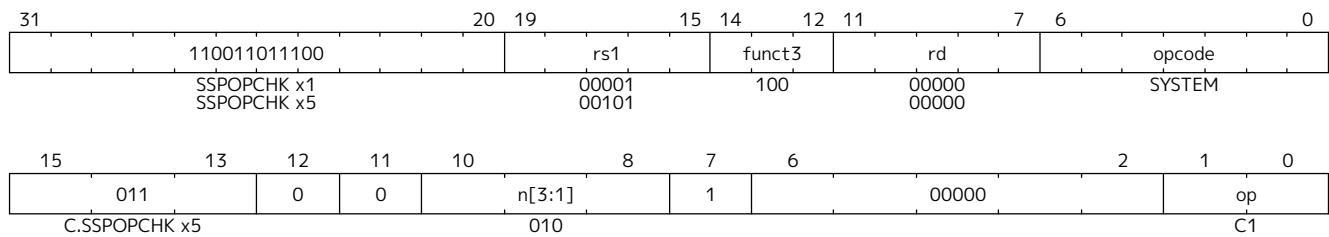
```
if (xsSE == 1)
    mem[ssp - (XLEN/8)] = X(src) # Store src value to ssp - XLEN/8
    ssp = ssp - (XLEN/8)         # decrement ssp by XLEN/8
endif
```

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.

32.2.5. 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.



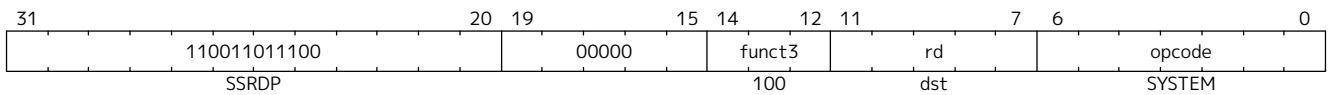
```
function_entry:
    c.addi sp,sp,-8 # push link register x1
    c.sd x1,(sp)    # on regular stack
    c.sspush x1     # push link register x1 on shadow stack
    :
    c.ld x5,(sp)    # pop link register x5 from regular
stack
    c.addi sp,sp,8
    c.sspopchk x5   # fault if x5 not equal to shadow 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`.

32.2.6. 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).

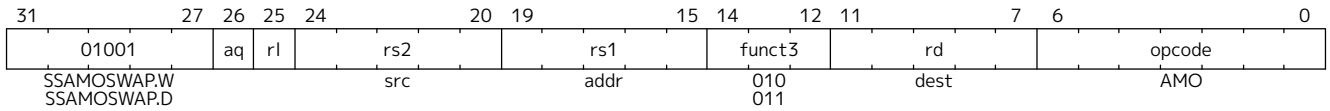
```

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("csw ssp, %0" : : "r" (cur_ssp));
    // Test if unwound past the shadow stack bounds
    asm("sspush x5");
    asm("sspopchk x5");
}
back_cfi_not_active:
:
}

```

32.2.7. 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 33. "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**.

33.1. 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.

33.2. 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

33.3. 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**.

33.4. 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.

33.5. Instructions

33.5.1. ld

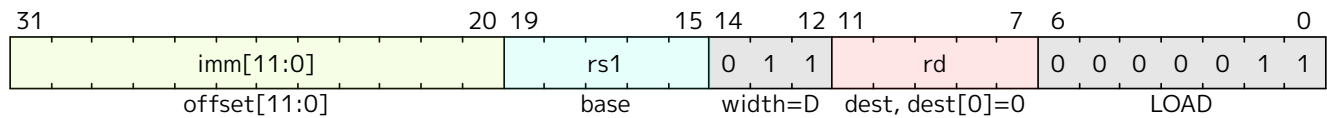
Synopsis

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: [Section 33.1](#)

DRAFT

33.5.2. sd

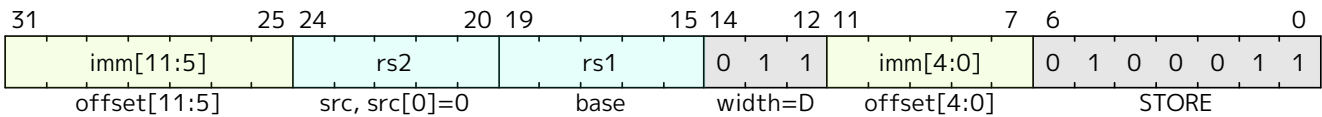
Synopsis

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: [Section 33.1](#)

DRAFT

33.5.3. c.ldsp

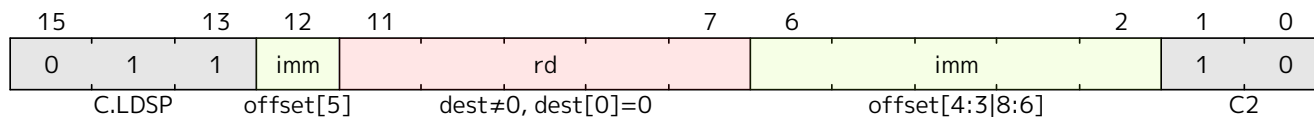
Synopsis

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 $rd \neq x0$; the code points with $rd = x0$ are reserved.

Included in: [Section 33.2](#)

DRAFT

33.5.4. c.sdsp

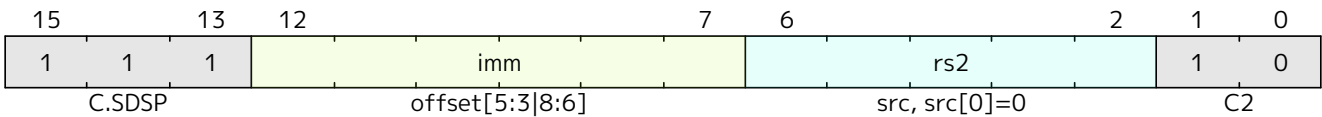
Synopsis

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: [Section 33.2](#)

DRAFT

33.5.5. c.ld

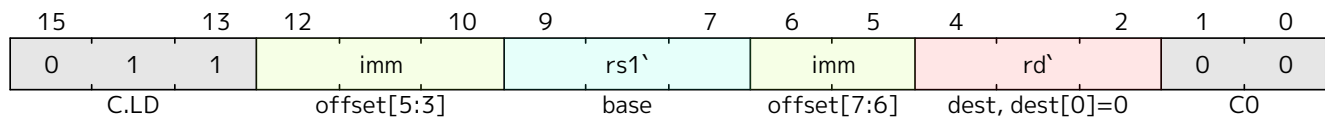
Synopsis

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: [Section 33.2](#)

DRAFT

33.5.6. c.sd

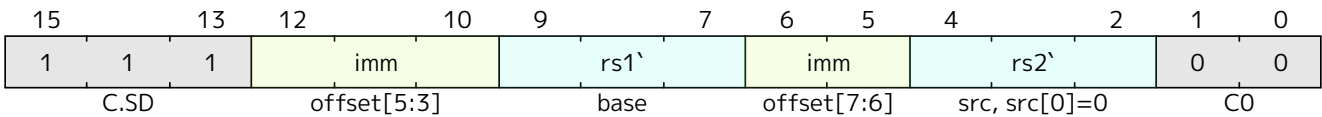
Synopsis

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: [Section 33.2](#)

DRAFT

Chapter 34. 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 73. 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	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	<i>reserved</i>
01		STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	<i>reserved</i>
10		MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	<i>custom-2</i>	<i>reserved</i>
11		BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	OP-VE	<i>custom-3</i>	<i>reserved</i>

Table 73 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 25 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
0000000000000			00000	000	00000	1110011	ECALL
0000000000001			00000	000	00000	1110011	EBREAK

DRAFT

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]	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

DRAFT

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

DRAFT

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

DRAFT

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	
000000001101	00000	000	00000	1110011	WRS.STO	

Table 74 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 74. 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 35. 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.

35.1. Case Sensitivity

The ISA naming strings are case insensitive.

35.2. 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.

35.3. 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.

35.4. Underscores

Underscores "_" may be used to separate ISA extensions to improve readability and to provide disambiguation, e.g., "RV32I2_M2_A2".

35.5. 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 [Section 5.1](#).

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".

35.6. 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.

35.7. 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.

35.8. 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.

35.9. 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.

35.10. 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".

35.11. Subset Naming Convention

Table 75 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 75. 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	

DRAFT

Appendix A: 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.

A.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.)

A.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 similar 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.

A.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.

A.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.

Appendix B: RVWMO Explanatory Material, Version 0.1

This section provides more explanation for RVWMO [Chapter 15](#), 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 15](#) and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in [Appendix B.7](#). Any other discrepancies are unintentional.

B.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.

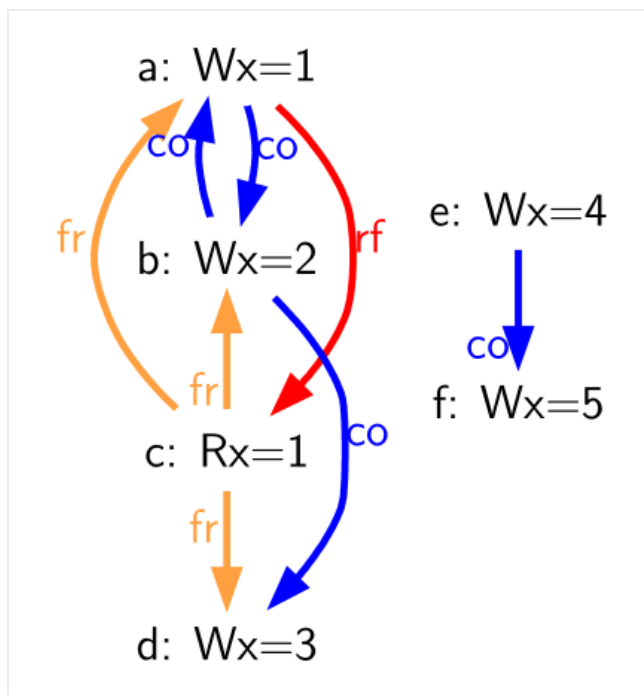
Some fences and/or memory ordering annotations in code written for RVWMO may become redundant under RVTSO; the cost that the default of RVWMO imposes on Ztso implementations is the incremental overhead of fetching those fences (e.g., FENCE R,RW and FENCE RW,W) which become no-ops on that implementation. However, these fences must remain present in the code if compatibility with non-Ztso implementations is desired.

B.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 76. 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 77. 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 [Appendix C.2](#). This notation is explained in [Table 77](#). 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.

B.3. Explaining the RVWMO Rules

In this section, we provide explanation and examples for all of the RVWMO rules and axioms.

B.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.

B.3.2. Load value axiom

Section 15.1.4.1: 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:

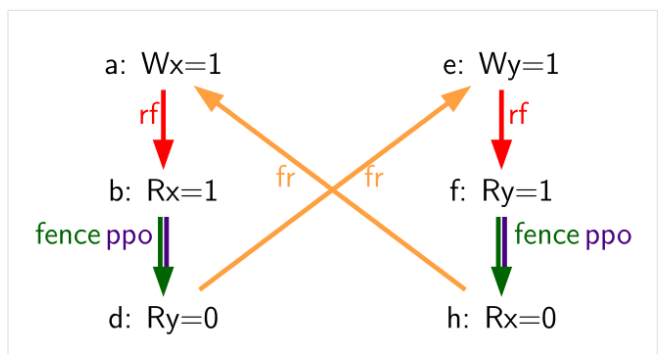
1. Stores that write that byte and that precede i in the global memory order
2. Stores that write that byte and that precede i in program order

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 78](#). 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 78. 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:

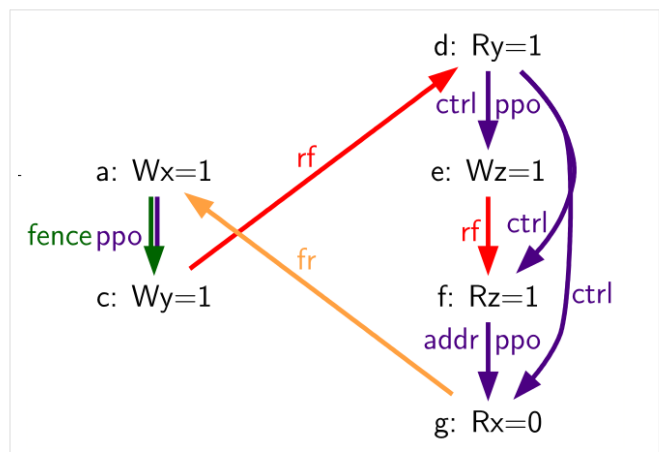
- (a) precedes (b): by rule X
- (b) precedes (d): by rule 4
- (d) precedes (e): by the load value axiom. Otherwise, if (e) preceded (d), then (d) would be required to return the value 1. (This is a perfectly legal execution; it's just not the one in question)
- (e) precedes (f): by rule X
- (f) precedes (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 79. 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 79. 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

B.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 80. 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 80 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 80](#) 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.

B.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.

B.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*

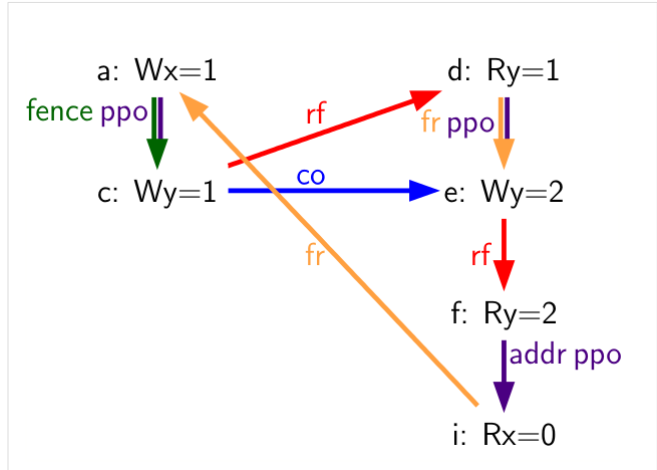
Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation 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 [Appendix B.3.2](#), 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 81. 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 81, 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:

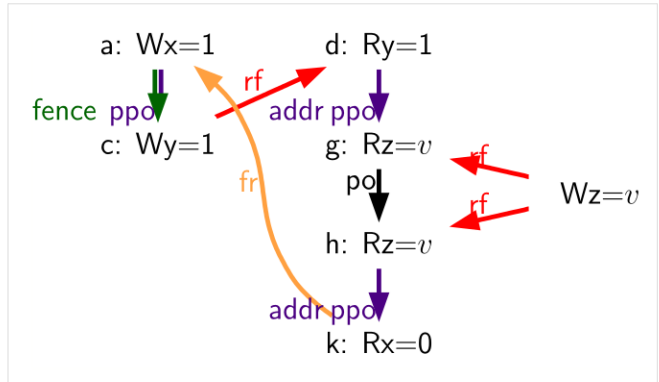
- (d) stalls (for whatever reason; perhaps it's stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer (but does not yet drain to memory)
- (f) executes and forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (a) executes and drains to memory, (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 82. 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,a 0
(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,a 2
		(j)	add s0,s0,t 3
		(k)	lw a3,0(s 0)
Outcome: a0=1, a1=v, a2=v, a3=0			



Consider the litmus test of Table 82. 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 82 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.

B.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.

B.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.

```

1      sd      x1, (a1)      # Arbitrary unrelated store
2      ld      x2, (a2)      # Arbitrary unrelated load
3      li      t0, 1         # Initialize swap value.
4      again:
5          amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
6          bnez      t0, again      # Retry if held.
7          # ...
8          # Critical section.
9          # ...
10         amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
11         sd      x3, (a3)      # Arbitrary unrelated store
12         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.

```

1      sd      x1, (a1)      # Arbitrary unrelated store
2      ld      x2, (a2)      # Arbitrary unrelated load
3      li      t0, 1         # Initialize swap value.
4      again:
5          amoswap.w      t0, t0, (a0) # Attempt to acquire lock.
6          fence          r, rw      # Enforce "acquire" memory ordering
7          bnez      t0, again      # Retry if held.
8          # ...
9          # Critical section.
10         # ...
11         fence          rw, w      # Enforce "release" memory ordering
12         amoswap.w      x0, x0, (a0) # Release lock by storing 0.
13         sd      x3, (a3)      # Arbitrary unrelated store
14         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.

B.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 15.1.2](#) 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)
```

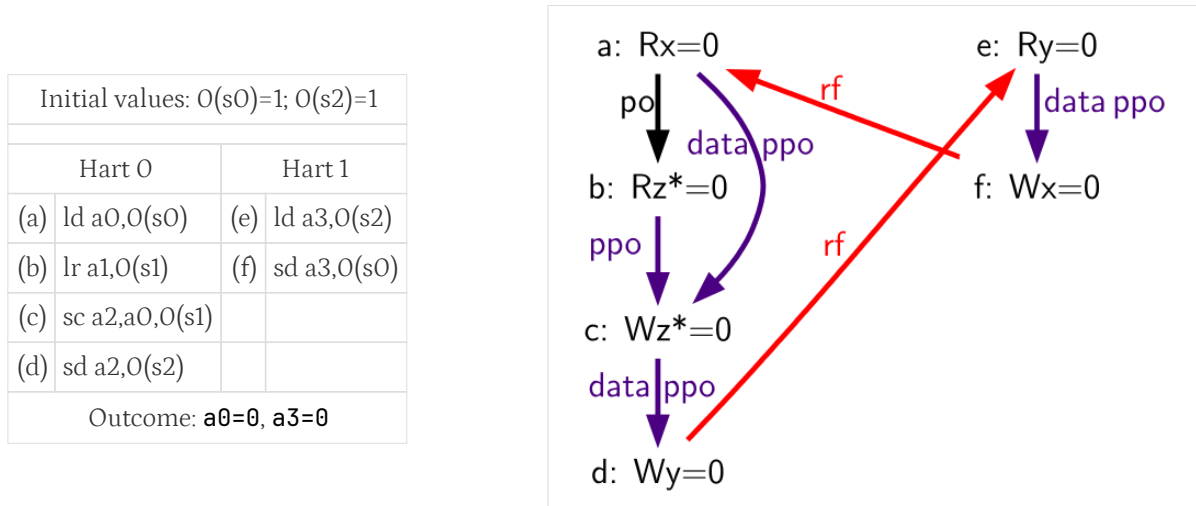
Listing 16. Another syntactic control dependency

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 83. A variant of the LB litmus test (outcome forbidden)



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 83. 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 15.3 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.

B.3.9. Pipeline Dependencies (Rules 12-13)

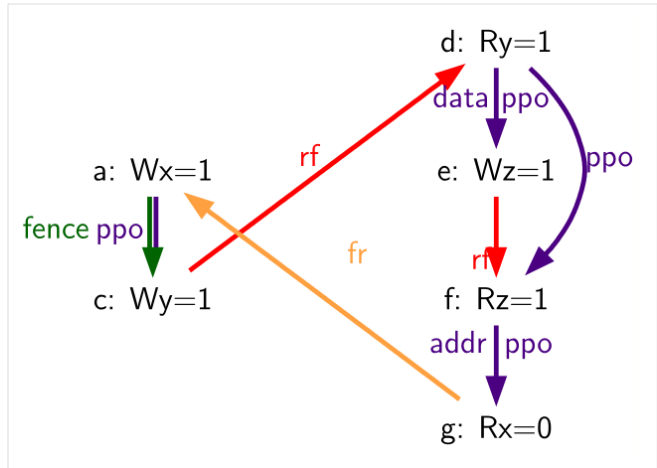


Rule 12: *b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m*

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 84. 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)

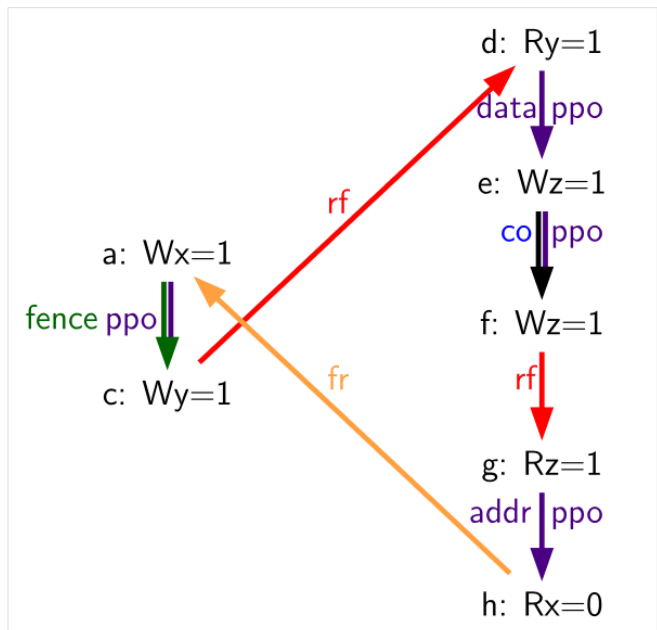
Hart 0	Hart 1
li t1, 1	(d) lw a0, 0(s1)
(a) sw t1, 0(s0)	(e) sw a0, 0(s2)
(b) fence w, w	(f) lw a1, 0(s2)
(c) sw t1, 0(s1)	xor a2, a1, a1
	add s0, s0, a2
	(g) lw a3, 0(s0)
Outcome: a0=1, a3=0	



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 84 (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 85. 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	

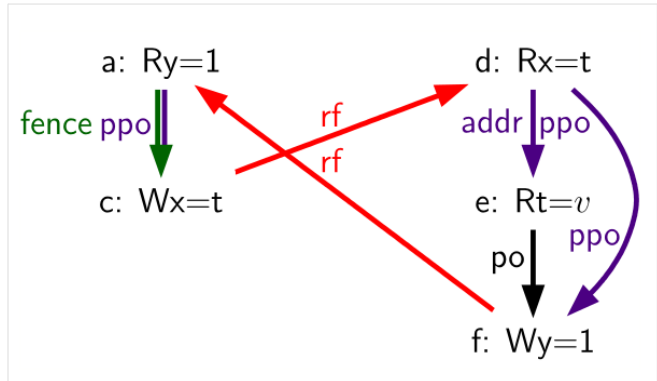


If there were another store to the same address in between (e) and (f), as in Table 86, then (f) would no longer be dependent on the data of (e) being resolved, and hence the dependency of (f) on (d), which produces the data for (e), would be broken.

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 86. 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 86 (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.

B.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 Section 5.1, 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.

B.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.

B.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 15](#), 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.

B.5. Code Porting and Mapping Guidelines

Table 87. Mappings from TSO operations to RISC-V operations

x86/TSO Operation	RVWMO Mapping
Load	<code>l{b h w d}; fence r,rw</code>
Store	<code>fence rw,w; s{b h w d}</code>
Atomic RMW	<code>amo<op>.{w d}.aqr l OR loop:lr.{w d}.aq; <op>; sc.{w d}.aqr l; bnez loop</code>
Fence	<code>fence rw,rw</code>

Table 87 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 87 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 88. Mappings from Power operations to RISC-V operations

Power Operation	RVWMO Mapping
Load	<code>l{b h w d}</code>
Load-Reserve	<code>lr.{w d}</code>
Store	<code>s{b h w d}</code>
Store-Conditional	<code>sc.{w d}</code>
<code>lwsync</code>	<code>fence.tso</code>
<code>sync</code>	<code>fence rw,rw</code>
<code>isync</code>	<code>fence.i; fence r,r</code>

Table 88 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 89. Mappings from ARM operations to RISC-V operations

ARM Operation	RVWMO Mapping
Load	<code>l{b h w d}</code>
Load-Acquire	<code>fence rw, rw; l{b h w d}; fence r, rw</code>
Load-Exclusive	<code>lr.{w d}</code>
Load-Acquire-Exclusive	<code>lr.{w d}.aqr l</code>
Store	<code>s{b h w d}</code>
Store-Release	<code>fence rw, w; s{b h w d}</code>
Store-Exclusive	<code>sc.{w d}</code>
Store-Release-Exclusive	<code>sc.{w d}.r l</code>
<code>dmb</code>	<code>fence rw, rw</code>
<code>dmb.ld</code>	<code>fence r, rw</code>
<code>dmb.st</code>	<code>fence w, w</code>
<code>isb</code>	<code>fence.i; fence r, r</code>

Table 89 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 89 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 90. Mappings from Linux memory primitives to RISC-V primitives.

Linux Operation	RVWMO Mapping
<code>smp_mb()</code>	<code>fence rw, rw</code>
<code>smp_rmb()</code>	<code>fence r, r</code>
<code>smp_wmb()</code>	<code>fence w, w</code>
<code>dma_rmb()</code>	<code>fence r, r</code>
<code>dma_wmb()</code>	<code>fence w, w</code>
<code>mb()</code>	<code>fence iorw, iorw</code>
<code>rmb()</code>	<code>fence ri, ri</code>
<code>wmb()</code>	<code>fence wo, wo</code>
<code>smp_load_acquire()</code>	<code>l{b h w d}; fence r, rw</code>
<code>smp_store_release()</code>	<code>fence.tso; s{b h w d}</code>
Linux Construct	RVWMO AMO Mapping
<code>atomic <op> relaxed</code>	<code>amo <op>.{w d}</code>
<code>atomic <op> acquire</code>	<code>amo <op>.{w d}.aq</code>
<code>atomic <op> release</code>	<code>amo <op>.{w d}.r l</code>

Linux Operation	RVWMO Mapping
atomic <op>	amo <op>.{w d}.aqr_l
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}.aqr_l*; 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}.aqr_l; bnez loop

With regards to [Table 90](#), 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 90](#) 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 90](#).

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.

```

1 Linux Code:
2 (a) int r0 = *x;
3     (bc) spin_unlock(y, 0);
4 ....
5 ....
6 (d) spin_lock(y);
```

```

7 (e) int r1 = *z;
8
9 RVWMO Mapping:
10 (a) lw a0, 0(s0)
11 (b) fence.tso // vs. fence rw,w
12 (c) sd x0,0(s1)
13 ....
14 loop:
15 (d) lr.d.aq a1,(s1)
16 bnez a1,loop
17 sc.d a1,t1,(s1)
18 bnez a1,loop
19 (e) lw a2,0(s2)

```

Listing 18. Orderings between critical sections in Linux

Table 91 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 92 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 91 sequentially consistent store, followed by a Table 92 sequentially consistent load can be reordered unless the Table 91 mapping of stores is strengthened by either adding a second fence or mapping the store to `amoswap.rl` instead.

Table 91. Mappings from C/C++ primitives to RISC-V primitives.

C/C++ Construct	RVWMO Mapping
Non-atomic load	<code>l{b h w d}</code>
<code>atomic_load(memory_order_relaxed)</code>	<code>l{b h w d}</code>
<code>atomic_load(memory_order_acquire)</code>	<code>l{b h w d}; fence r,rw</code>
<code>atomic_load(memory_order_seq_cst)</code>	<code>fence rw,rw; l{b h w d}; fence r,rw</code>
Non-atomic store	<code>s{b h w d}</code>
<code>atomic_store(memory_order_relaxed)</code>	<code>s{b h w d}</code>
<code>atomic_store(memory_order_release)</code>	<code>fence rw,w; s{b h w d}</code>
<code>atomic_store(memory_order_seq_cst)</code>	<code>fence rw,w; s{b h w d}</code>
<code>atomic_thread_fence(memory_order_acquire)</code>	<code>fence r,rw</code>
<code>atomic_thread_fence(memory_order_release)</code>	<code>fence rw,w</code>
<code>atomic_thread_fence(memory_order_acq_rel)</code>	<code>fence.tso</code>
<code>atomic_thread_fence(memory_order_seq_cst)</code>	<code>fence rw,rw</code>
C/C++ Construct	RVWMO AMO Mapping
<code>atomic_<op>(memory_order_relaxed)</code>	<code>amo<op>.{w d}</code>
<code>atomic_<op>(memory_order_acquire)</code>	<code>amo<op>.{w d}.aq</code>
<code>atomic_<op>(memory_order_release)</code>	<code>amo<op>.{w d}.rl</code>
<code>atomic_<op>(memory_order_acq_rel)</code>	<code>amo<op>.{w d}.aqr</code>
<code>atomic_<op>(memory_order_seq_cst)</code>	<code>amo<op>.{w d}.aqr</code>
C/C++ Construct	RVWMO LR/SC Mapping
<code>atomic_<op>(memory_order_relaxed)</code>	<code>loop:lr.{w d}; <op>; sc.{w d};</code>

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}.aqr; <op>;
	sc.{w d}.rl; bnez loop

Table 92. 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}.aqr
atomic_<op>(memory_order_seq_cst)	amo<op>.{w d}.aqr
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}.aqr in order to interoperate with code mapped per Table 91	

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
- RsrEventual

Platforms with different attributes may require different mappings, or require platform-specific SW (e.g., memory-mapped I/O).

B.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 [Appendix C.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 [Appendix B.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

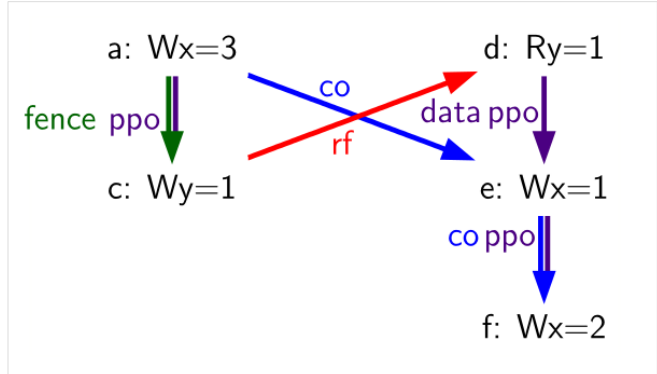
cases such as RSW [Appendix B.3.5](#) which tend to be incompatible with silent stores.

- 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 93. 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).

B.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

B.7. Known Issues

B.7.1. Mixed-size RSW

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)	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 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)	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 96. 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 94-Table 96](#). 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.

DRAFT

Appendix C: 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 15](#) and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in [Appendix B.7](#). Any other discrepancies are unintentional.

C.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 [Appendix B.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 :> FenceTS0).( ^po :> MemoryEvent)
+ (Store <: ^po :> FenceTS0).( ^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
  + FenceTS0
  + 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)

C.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 15](#), 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)

C.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/rem-s-project/rmem](https://github.com/REMS-project/rmem)). `rmem` can explore litmus tests (see [Appendix B.2](#)) and small ELF binaries exhaustively, pseudorandomly and interactively. In `rmem`, the ISA semantics is expressed explicitly in Sail (see [github.com/rem-s-project/sail](https://github.com/REMS-project/sail) for the Sail language, and [github.com/rem-s-project/sail-riscv](https://github.com/REMS-project/sail-riscv) for the RISC-V ISA model), and the concurrency semantics is expressed in Lem (see [github.com/rem-s-project/lem](https://github.com/REMS-project/lem) for the Lem language).

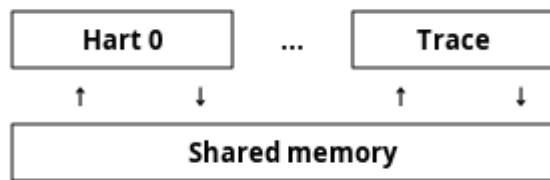
rmem has a command-line interface and a web-interface. The web-interface runs entirely on the client side, and is provided online together with a library of litmus tests: www.cl.cam.ac.uk/. The command-line interface is faster than the web-interface, specially in exhaustive mode.

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 [Appendix C.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 unpropagated 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 $\bigcirc$ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behavior; the $\bullet$ cannot. Although **Fetch instruction** is marked with a $\bullet$, 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.

C.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):

<code>Load_mem(kind, address, size, load_continuation)</code>	- memory load operation
<code>Early_sc_fail(res_continuation)</code>	- allow sc to fail early
<code>Store_ea(kind, address, size, next_state)</code>	- 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).

C.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(<i>isa_state</i>)	- ready to make a pseudocode transition
Pending_mem_loads(<i>load_continuation</i>)	- requesting memory load operation(s)
Pending_mem_stores(<i>store_continuation</i>)	- 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.*

C.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.

C.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.*

C.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.

C.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 (**j_{al}**), the target address;
- for an indirect jump instruction (**j_{alr}**), 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 (**j_{alr}**), 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.

C.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 15.1.1](#) 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.

C.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.

C.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 $msos$ 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 unpropagated stores](#).



Note that [Satisfy memory operation by forwarding from unpropagated 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.

C.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 $i.mem\_loads$.

C.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))`.

C.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))`.

C.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).

C.3.5.9. Instantiate memory store operation values

An instruction instance i in state $\text{Plain}(\text{Store\_memv}(\text{mem\_value}, \text{store\_continuation}))$ can always instantiate the values of the memory store operations $i.\text{mem\_stores}$. Action:

1. split mem_value between the memory store operations $i.\text{mem\_stores}$; and
2. update the state of i to $\text{Pending\_mem\_stores}(\text{store\_continuation})$.

C.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 $\text{Pending\_mem\_stores}(\text{store\_continuation})$, 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.

C.3.5.11. Propagate store operation

For a committed instruction instance i in state $\text{Pending\_mem\_stores}(\text{store\_continuation})$, and an unpropagated memory store operation mso in $i.\text{mem\_stores}$, 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](#)).

C.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](#).

C.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.

C.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))`.

C.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. [Complete 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.

C.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 $\text{Plain}(\text{next\_state})$.

C.3.5.17. Register read

An instruction instance i in state $\text{Plain}(\text{Read\_reg}(\text{reg\_name}, \text{read\_cont}))$ 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 $\text{initial\_register\_state}$. Let reg_value be the value assembled from read_sources .

Action:

1. add reg_name to $i.\text{reg\_reads}$ with read_sources and reg_value ; and
2. update the state of i to $\text{Plain}(\text{read\_cont}(\text{reg\_value}))$.

C.3.5.18. Register write

An instruction instance i in state $\text{Plain}(\text{Write\_reg}(\text{reg\_name}, \text{reg\_value}, \text{next\_state}))$ can always do a reg_name register write. Action:

1. add reg_name to $i.\text{reg\_writes}$ with deps and reg_value ; and
2. update the state of i to $\text{Plain}(\text{next\_state})$.

where deps is a pair of the set of all read_sources from $i.\text{reg\_reads}$, and a flag that is true iff i is a load instruction instance that has already been entirely satisfied.

C.3.5.19. Pseudocode internal step

An instruction instance i in state $\text{Plain}(\text{Internal}(\text{next\_state}))$ can always do that pseudocode-internal step. Action: update the state of i to $\text{Plain}(\text{next\_state})$.

C.3.5.20. Finish instruction

A non-finished instruction instance i in state $\text{Plain}(\text{Done})$ 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 $\neg \text{cfp}$ be the complement of cfp in the memory footprint of i . If $\neg \text{cfp}$ 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 determine 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*.

C.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 D: Vector Assembly Code Examples

The following are provided as non-normative text to help explain the vector ISA.

D.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
```

D.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?

```

D.3. Malloc 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

```

D.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

```

D.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
```

D.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
    vmacc.vf v2, ft2, v16
    flw ft6, (amp)
    add amp, amp, astride
    flw ft7, (amp)
    vmacc.vf v3, ft3, v16
    add amp, amp, astride
    flw ft8, (amp)
    add amp, amp, astride
    vmacc.vf v4, ft4, v16
    flw ft9, (amp)
    add amp, amp, astride
    vmacc.vf v5, ft5, v16
    flw ft10, (amp)
    add amp, amp, astride
    vmacc.vf v6, ft6, v16
    flw ft11, (amp)
    add amp, amp, astride
    vmacc.vf v7, ft7, v16
    flw ft12, (amp)
    add amp, amp, astride
    vmacc.vf v8, ft8, v16
    flw ft13, (amp)
    add amp, amp, astride
    vmacc.vf v9, ft9, v16
    flw ft14, (amp)
    add amp, amp, astride
    vmacc.vf v10, ft10, v16
    flw ft15, (amp)
    add amp, amp, astride
    addi akp, akp, 4           # Move to next column of a
    vmacc.vf v11, ft11, v16
    beqz kt, 1f               # Don't load past end of matrix
    flw ft0, (akp)
    add amp, akp, astride
1:  vmacc.vf v12, ft12, v16
    beqz kt, 1f
    flw ft1, (amp)
    add amp, amp, astride
1:  vmacc.vf v13, ft13, v16
    beqz kt, 1f
    flw ft2, (amp)
    add amp, amp, astride
1:  vmacc.vf v14, ft14, v16
    beqz kt, 1f             # Exit out of loop
    flw ft3, (amp)
    add amp, amp, astride
    vmacc.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, FRAME_SIZE
    ret

```

D.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)
vfmsub.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)
vfmsub.vv v3, v4, v3     # Better estimate of 1/v2
vfmul.vv v1, v1, v3      # Estimate of v1/v2

```

D.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

```


D.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

```

D.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
```

DRAFT

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

```


Part II: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture

DRAFT

Preface

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

Module	Version
Machine ISA	1.13
Smstateen Extension	1.0
Smcsrind/Sscsrind Extension	1.0
Smepmp Extension	1.0
Smcptrpmf Extension	1.0
Smrnmi Extension	1.0
Smcdeleg Extension	1.0
Smdbltrp Extension	1.0
Smctr Extension	1.0
Supervisor ISA	1.13
Svade Extension	1.0
Svnapot Extension	1.0
Svpbmt Extension	1.0
Svinval Extension	1.0
Svadu Extension	1.0
Svvptc Extension	1.0
Ssqosid Extension	1.0
Sstc Extension	1.0
Sscofpmf Extension	1.0
Ssdbltrp Extension	1.0
Ssqosid Extension	1.0
Hypervisor ISA	1.0
Shlcofideleg Extension	1.0
Svvptc Extension	1.0
Pointer-Masking Extensions	1.0
Svrsw60t59b Extension	1.0

The following changes have been made since version 20250508:

- Addition of the Svrsw60t59b extension for additional PTE reserved-for-software bits.

Preface to Version 20250508

This document describes the RISC-V privileged 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:

Module	Version	Status
Machine ISA	1.13	Ratified
Smstateen Extension	1.0	Ratified
Smcsrind/Sscsrind Extension	1.0	Ratified
Smepmp Extension	1.0	Ratified
Smcptrpmf Extension	1.0	Ratified
Smrnmi Extension	1.0	Ratified
Smcdeleg Extension	1.0	Ratified
Smdbltrp Extension	1.0	Ratified
Smctr Extension	1.0	Ratified
Supervisor ISA	1.13	Ratified
Svade Extension	1.0	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Svadu Extension	1.0	Ratified
Svvptc Extension	1.0	Ratified
Ssqosid Extension	1.0	Ratified
Sstc Extension	1.0	Ratified
Sscofpmf Extension	1.0	Ratified
Ssdbltrp Extension	1.0	Ratified
Ssqosid Extension	1.0	Ratified
Hypervisor ISA	1.0	Ratified
Shlcofideleg Extension	1.0	Ratified
Svvptc Extension	1.0	Ratified
Pointer-Masking Extensions	1.0	Ratified

The following changes have been made since version 20241101:

- Addition of the Smctr Control Transfer Records extension.
- Addition of the Svvpct Extension for Obviating Memory-Management Instructions after Marking PTEs Valid.
- Addition of the Ssqosid Extension for Quality-of-Service Identifiers.
- Addition of the Pointer-Masking Extensions.

Preface to Version 20241101

This document describes the RISC-V privileged 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:

Module	Version	Status
Machine ISA	1.13	Ratified
Smstateen Extension	1.0	Ratified
Smcsrind/Sscsrind Extension	1.0	Ratified
Smepmp Extension	1.0	Ratified
Smcptrpmf Extension	1.0	Ratified
Smrnmi Extension	1.0	Ratified
Smcdeleg Extension	1.0	Ratified
Smdbltrp Extension	1.0	Ratified
Supervisor ISA	1.13	Ratified
Svade Extension	1.0	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Svadu Extension	1.0	Ratified
Sstc Extension	1.0	Ratified
Sscofpmf Extension	1.0	Ratified
Ssdbltrp Extension	1.0	Ratified
Ssqosid Extension	1.0	Ratified
Hypervisor ISA	1.0	Ratified
Shlcofideleg Extension	1.0	Ratified
Svpptc Extension	1.0	Ratified

Preface to Version 20241017

This document describes the RISC-V privileged architecture. This release, version 20241017, contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	1.13	Ratified
Smstateen Extension	1.0	Ratified
Smcsrind/Sscsrind Extension	1.0	Ratified
Smepmp	1.0	Ratified
Smcptrpmf	1.0	Ratified
Smrnmi Extension	1.0	Ratified
Smcdeleg	1.0	Ratified
Smdbltrp	1.0	Ratified
Supervisor ISA	1.13	Ratified
Svade Extension	1.0	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Svadu Extension	1.0	Ratified
Sstc	1.0	Ratified
Sscofpmf	1.0	Ratified
Ssdbltrp	1.0	Ratified
Hypervisor ISA	1.0	Ratified
Shlcofideleg	1.0	Ratified
Svpptc	1.0	Ratified

The following changes have been made since version 1.12 of the Machine and Supervisor ISAs, which, while not strictly backwards compatible, are not anticipated to cause software portability problems in practice:

- Redefined **misal.MXL** to be read-only, making **MXLEN** a constant.
- Added the constraint that $SXLEN \geq UXLEN$.

Additionally, the following compatible changes have been made to the Machine and Supervisor ISAs since version 1.12:

- Defined the **misal.B** field to reflect that the B extension has been implemented.
- Defined the **misal.V** field to reflect that the V extension has been implemented.
- Defined the RV32-only **medeleg** and **hedeleg** CSRs.
- Defined the misaligned atomicity granule PMA, superseding the proposed Zam extension.
- Allocated interrupt 13 for Sscofpmf LCOFI interrupt.
- Defined hardware-error and software-check exception codes.
- Specified synchronization requirements when changing the PBMTE and ADUE fields in **menvcfg** and **henvcfg**.
- Exposed count-overflow interrupts to VS-mode via the **Shlcofideleg** extension.
- Relaxed behavior of some HINTs when $MXLEN > XLEN$.
- Defined the format of the memory-mapped **msip** registers.

Finally, the following clarifications and document improvements have been made since the last document release:

- Transliterated the document from LaTeX into AsciiDoc.
- Included all ratified extensions through March 2024.
- Clarified that "platform- or custom-use" interrupts are actually "platform-use interrupts", where the platform can choose to make some custom.
- Clarified semantics of explicit accesses to CSRs wider than $XLEN$ bits.
- Clarified that $MXLEN \geq SXLEN$.
- Clarified that WFI is not a HINT instruction.
- Clarified that VS-stage page-table accesses set G-stage A/D bits.
- Clarified ordering rules when $PBMT=IO$ is used on main-memory regions.
- Clarified ordering rules for hardware A/D bit updates.
- Clarified that, for a given exception cause, **xtval** might sometimes be set to a nonzero value but sometimes not.
- Clarified exception behavior of unimplemented or inaccessible CSRs.
- Clarified that **Svpbmt** allows implementations to override additional PMAs.
- Replaced the concept of vacant memory regions with inaccessible memory or I/O regions.
- Clarified that timer and count-overflow interrupts' arrival in interrupt-pending registers is not immediate.
- Clarified that **MXR** affects only explicit memory accesses.

Preface to Version 20211203

This document describes the RISC-V privileged architecture. This release, version 20211203, contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	1.12	Ratified
Supervisor ISA	1.12	Ratified
Svnapot Extension	1.0	Ratified
Svpbmt Extension	1.0	Ratified
Svinval Extension	1.0	Ratified
Hypervisor ISA	1.0	Ratified

The following changes have been made since version 1.11, which, while not strictly backwards compatible, are not anticipated to cause software portability problems in practice:

- Changed MRET and SRET to clear **mstatus.MPRV** when leaving M-mode.
- Reserved additional **satp** patterns for future use.
- Stated that the **scause** Exception Code field must implement bits 4–0 at minimum.
- Relaxed I/O regions have been specified to follow RVWMO. The previous specification implied that PPO rules other than fences and acquire/release annotations did not apply.
- Constrained the LR/SC reservation set size and shape when using page-based virtual memory.
- PMP changes require an SFENCE.VMA on any hart that implements page-based virtual memory, even if VM is not currently enabled.
- Allowed for speculative updates of page table entry A bits.
- Clarify that if the address-translation algorithm non-speculatively reaches a PTE in which a bit reserved for future standard use is set, a page-fault exception must be raised.

Additionally, the following compatible changes have been made since version 1.11:

- Removed the N extension.
- Defined the mandatory RV32-only CSR **mstatush**, which contains most of the same fields as the upper 32 bits of RV64's **mstatus**.
- Defined the mandatory CSR **mconfigptr**, which if nonzero contains the address of a configuration data structure.
- Defined **mseccfg** and **mseccfgh** CSRs, which control the machine's security configuration.
- Defined **menvcfg**, **henvcfg**, and **senvcfg** CSRs (and RV32-only **menvcfgh** and **henvcfgh** CSRs), which control various characteristics of the execution environment.
- Designated part of SYSTEM major opcode for custom use.
- Permitted the unconditional delegation of less-privileged interrupts.
- Added optional big-endian and bi-endian support.
- Made priority of load/store/AMO address-misaligned exceptions implementation-defined relative to load/store/AMO page-fault and access-fault exceptions.
- PMP reset values are now platform-defined.
- An additional 48 optional PMP registers have been defined.
- Slightly relaxed the atomicity requirement for A and D bit updates performed by the implementation.
- Clarify the architectural behavior of address-translation caches

- Added Sv57 and Sv57x4 address translation modes.
- Software breakpoint exceptions are permitted to write either 0 or the `pc` to `xtval`.
- Clarified that bare S-mode need not support the SFENCE.VMA instruction.
- Specified relaxed constraints for implicit reads of non-idempotent regions.
- Added the Svnepot Standard Extension, along with the N bit in Sv39, Sv48, and Sv57 PTEs.
- Added the Svpbmt Standard Extension, along with the PBMT bits in Sv39, Sv48, and Sv57 PTEs.
- Added the Svinval Standard Extension and associated instructions.

Finally, the hypervisor architecture proposal has been extensively revised.

Preface to Version 1.11

This is version 1.11 of the RISC-V privileged architecture. The document contains the following versions of the RISC-V ISA modules:

Module	Version	Status
Machine ISA	1.11	Ratified
Supervisor ISA	1.11	Ratified
Hypervisor ISA	0.3	Draft

Changes from version 1.10 include:

- Moved Machine and Supervisor spec to **Ratified** status.
- Improvements to the description and commentary.
- Added a draft proposal for a hypervisor extension.
- Specified which interrupt sources are reserved for standard use.
- Allocated some synchronous exception causes for custom use.
- Specified the priority ordering of synchronous exceptions.
- Added specification that xRET instructions may, but are not required to, clear LR reservations if A extension present.
- The virtual-memory system no longer permits supervisor mode to execute instructions from user pages, regardless of the SUM setting.
- Clarified that ASIDs are private to a hart, and added commentary about the possibility of a future global-ASID extension.
- SFENCE.VMA semantics have been clarified.
- Made the `mstatus.MPP` field **WARL**, rather than **WLRL**.
- Made the unused `xip` fields **WPRI**, rather than **WIRI**.
- Made the unused `mis` fields **WARL**, rather than **WIRI**.
- Made the unused `pmpaddr` and `pmpcfg` fields **WARL**, rather than **WIRI**.
- Required all harts in a system to employ the same PTE-update scheme as each other.
- Rectified an editing error that misdescribed the mechanism by which `mstatus.xIE` is written upon an exception.
- Described scheme for emulating misaligned AMOs.
- Specified the behavior of the `mis` and `xepc` registers in systems with variable IALIGN.

- Specified the behavior of writing self-contradictory values to the **misa** register.
- Defined the **mcountinhibit** CSR, which stops performance counters from incrementing to reduce energy consumption.
- Specified semantics for PMP regions coarser than four bytes.
- Specified contents of CSRs across XLEN modification.
- Moved PLIC chapter into its own document.

Preface to Version 1.10

This is version 1.10 of the RISC-V privileged architecture proposal. Changes from version 1.9.1 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.
- The explicit convention on shadow CSR addresses has been removed to reclaim CSR space. Shadow CSRs can still be added as needed.
- The **mvendorid** register now contains the JEDEC code of the core provider as opposed to a code supplied by the Foundation. This avoids redundancy and offloads work from the Foundation.
- The interrupt-enable stack discipline has been simplified.
- An optional mechanism to change the base ISA used by supervisor and user modes has been added to the **mstatus** CSR, and the field previously called Base in **misa** has been renamed to **MXL** for consistency.
- Clarified expected use of XS to summarize additional extension state status fields in **mstatus**.
- Optional vectored interrupt support has been added to the **mtvec** and **stvec** CSRs.
- The SEIP and UEIP bits in the **mip** CSR have been redefined to support software injection of external interrupts.
- The **mbadaddr** register has been subsumed by a more general **mtval** register that can now capture bad instruction bits on an illegal-instruction fault to speed instruction emulation.
- The machine-mode base-and-bounds translation and protection schemes have been removed from the specification as part of moving the virtual memory configuration to **sptbr** (now **satp**). Some of the motivation for the base and bound schemes are now covered by the PMP registers, but space remains available in **mstatus** to add these back at a later date if deemed useful.
- In systems with only M-mode, or with both M-mode and U-mode but without U-mode trap support, the **medeleg** and **mideleg** registers now do not exist, whereas previously they returned zero.
- Virtual-memory page faults now have **mcause** values distinct from physical-memory access faults. Page-fault exceptions can now be delegated to S-mode without delegating exceptions generated by PMA and PMP checks.
- An optional physical-memory protection (PMP) scheme has been proposed.
- The supervisor virtual memory configuration has been moved from the **mstatus** register to the **sptbr** register. Accordingly, the **sptbr** register has been renamed to **satp** (Supervisor Address Translation and Protection) to reflect its broadened role.
- The SFENCE.VM instruction has been removed in favor of the improved SFENCE.VMA instruction.
- The **mstatus** bit MXR has been exposed to S-mode via **sstatus**.
- The polarity of the PUM bit in **sstatus** has been inverted to shorten code sequences involving MXR. The bit has been renamed to SUM.
- Hardware management of page-table entry Accessed and Dirty bits has been made optional; simpler

implementations may trap to software to set them.

- The counter-enable scheme has changed, so that S-mode can control availability of counters to U-mode.
- H-mode has been removed, as we are focusing on recursive virtualization support in S-mode. The encoding space has been reserved and may be repurposed at a later date.
- A mechanism to improve virtualization performance by trapping S-mode virtual-memory management operations has been added.
- The Supervisor Binary Interface (SBI) chapter has been removed, so that it can be maintained as a separate specification.

Preface to Version 1.9.1

This is version 1.9.1 of the RISC-V privileged architecture proposal. Changes from version 1.9 include:

- Numerous additions and improvements to the commentary sections.
- Change configuration string proposal to be use a search process that supports various formats including Device Tree String and flattened Device Tree.
- Made **mis**a optionally writable to support modifying base and supported ISA extensions. CSR address of **mis**a changed.
- Added description of debug mode and debug CSRs.
- Added a hardware performance monitoring scheme. Simplified the handling of existing hardware counters, removing privileged versions of the counters and the corresponding delta registers.
- Fixed description of SPIE in presence of user-level interrupts.

Chapter 1. Introduction

This volume describes the RISC-V privileged architecture, which covers all aspects of RISC-V systems beyond the unprivileged ISA, including privileged instructions as well as additional functionality required for running operating systems and attaching external devices.

Commentary on our design decisions is formatted as in this paragraph, and can be skipped if the reader is only interested in the specification itself.



We briefly note that the entire privileged-level design described in this volume could be replaced with an entirely different privileged-level design without changing the unprivileged ISA, and possibly without even changing the ABI. In particular, this privileged specification was designed to run existing popular operating systems, and so embodies the conventional level-based protection model. Alternate privileged specifications could embody other more flexible protection-domain models. For simplicity of expression, the text is written as if this was the only possible privileged architecture.

1.1. RISC-V Privileged Software Stack Terminology

This section describes the terminology we use to describe components of the wide range of possible privileged software stacks for RISC-V.

Figure 8 shows some of the possible software stacks that can be supported by the RISC-V architecture. The left-hand side shows a simple system that supports only a single application running on an application execution environment (AEE). The application is coded to run with a particular application binary interface (ABI). The ABI includes the supported user-level ISA plus a set of ABI calls to interact with the AEE. The ABI hides details of the AEE from the application to allow greater flexibility in implementing the AEE. The same ABI could be implemented natively on multiple different host OSs, or could be supported by a user-mode emulation environment running on a machine with a different native ISA.



Our graphical convention represents abstract interfaces using black boxes with white text, to separate them from concrete instances of components implementing the interfaces.

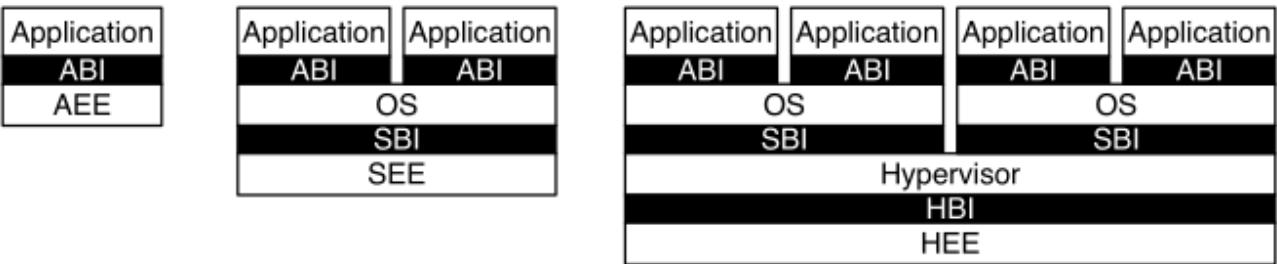


Figure 8. Different implementation stacks supporting various forms of privileged execution.

The middle configuration shows a conventional operating system (OS) that can support multiprogrammed execution of multiple applications. Each application communicates over an ABI with the OS, which provides the AEE. Just as applications interface with an AEE via an ABI, RISC-V operating systems interface with a supervisor execution environment (SEE) via a supervisor binary interface (SBI). An SBI comprises the user-level and supervisor-level ISA together with a set of SBI function calls. Using a single SBI across all SEE implementations allows a single OS binary image to run on any SEE. The SEE can be a simple boot loader and BIOS-style IO system in a low-end hardware platform, or a hypervisor-provided virtual machine in a high-end server, or a thin translation layer over a host operating system in an

architecture simulation environment.



Most supervisor-level ISA definitions do not separate the SBI from the execution environment and/or the hardware platform, complicating virtualization and bring-up of new hardware platforms.

The rightmost configuration shows a virtual machine monitor configuration where multiple multiprogrammed OSs are supported by a single hypervisor. Each OS communicates via an SBI with the hypervisor, which provides the SEE. The hypervisor communicates with the hypervisor execution environment (HEE) using a hypervisor binary interface (HBI), to isolate the hypervisor from details of the hardware platform.



The ABI, SBI, and HBI are still a work-in-progress, but we are now prioritizing support for Type-2 hypervisors where the SBI is provided recursively by an S-mode OS.

Hardware implementations of the RISC-V ISA will generally require additional features beyond the privileged ISA to support the various execution environments (AEE, SEE, or HEE).

1.2. Privilege Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined as shown in [Table 97](#).

Table 97. RISC-V privilege levels.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

Privilege levels are used to provide protection between different components of the software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment.



In the description, we try to separate the privilege level for which code is written, from the privilege mode in which it runs, although the two are often tied. For example, a supervisor-level operating system can run in supervisor-mode on a system with three privilege modes, but can also run in user-mode under a classic virtual machine monitor on systems with two or more privilege modes. In both cases, the same supervisor-level operating system binary code can be used, coded to a supervisor-level SBI and hence expecting to be able to use supervisor-level privileged instructions and CSRs. When running a guest OS in user mode, all supervisor-level actions will be trapped and emulated by the SEE running in the higher-privilege level.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode can be used to manage secure execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage respectively.

Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports an optional standard extension for memory protection. Also, supervisor mode can be extended to support Type-2 hypervisor execution as described in [Chapter 5](#).

Implementations might provide anywhere from 1 to 3 privilege modes trading off reduced isolation for lower implementation cost, as shown in [Table 98](#).

Table 98. Supported combination of privilege modes.

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Systems running Unix-like operating systems

All hardware implementations must provide M-mode, as this is the only mode that has unfettered access to the whole machine. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code.



The lock feature of the optional PMP facility can provide some limited protection even with only M-mode implemented.

Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system from application code. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE.

A hart normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Traps that increase privilege level are termed *vertical* traps, while traps that remain at the same privilege level are termed *horizontal* traps. The RISC-V privileged architecture provides flexible routing of traps to different privilege layers.



Horizontal traps can be implemented as vertical traps that return control to a horizontal trap handler in the less-privileged mode.

1.3. Debug Mode

Implementations may also include a debug mode to support off-chip debugging and/or manufacturing test. Debug mode (D-mode) can be considered an additional privilege mode, with even more access than M-mode. The separate debug specification describes operation of a RISC-V hart in debug mode. Debug mode reserves a few CSR addresses that are only accessible in D-mode, and may also reserve some portions of the physical address space on a platform.

Chapter 2. Control and Status Registers (CSRs)

The SYSTEM major opcode is used to encode all privileged instructions in the RISC-V ISA. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), which are defined in the Zicsr extension, and all other privileged instructions. The privileged architecture requires the Zicsr extension; which other privileged instructions are required depends on the privileged-architecture feature set.

In addition to the unprivileged state described in Volume I of this manual, an implementation may contain additional CSRs, accessible by some subset of the privilege levels using the CSR instructions described in [Volume I, Chapter 6](#). In this chapter, we map out the CSR address space. The following chapters describe the function of each of the CSRs according to privilege level, as well as the other privileged instructions which are generally closely associated with a particular privilege level. Note that although CSRs and instructions are associated with one privilege level, they are also accessible at all higher privilege levels.

Standard CSRs do not have side effects on reads but may have side effects on writes.

2.1. CSR Address Mapping Conventions

The standard RISC-V ISA sets aside a 12-bit encoding space (`csr[11:0]`) for up to 4,096 CSRs. By convention, the upper 4 bits of the CSR address (`csr[11:8]`) are used to encode the read and write accessibility of the CSRs according to privilege level as shown in [Table 99](#). The top two bits (`csr[11:10]`) indicate whether the register is read/write (**00**, **01**, or **10**) or read-only (**11**). The next two bits (`csr[9:8]`) encode the lowest privilege level that can access the CSR, with the pattern **10** representing hypervisor CSRs.



The CSR address convention uses the upper bits of the CSR address to encode default access privileges. This simplifies error checking in the hardware and provides a larger CSR space, but does constrain the mapping of CSRs into the address space.

Implementations might allow a more-privileged level to trap otherwise permitted CSR accesses by a less-privileged level to allow these accesses to be intercepted. This change should be transparent to the less-privileged software.

Instructions that access a non-existent CSR are reserved. Attempts to access a CSR without appropriate privilege level raise illegal-instruction exceptions or, as described in [Section 5.6.1](#), virtual-instruction exceptions. Attempts to write a read-only register raise illegal-instruction exceptions. A read/write register might also contain some bits that are read-only, in which case writes to the read-only bits are ignored.

[Table 99](#) also indicates the convention to allocate CSR addresses between standard and custom uses. The CSR addresses designated for custom uses will not be redefined by future standard extensions.

Machine-mode standard read-write CSRs **0x7A0-0x7BF** are reserved for use by the debug system. Of these CSRs, **0x7A0-0x7AF** are accessible to machine mode, whereas **0x7B0-0x7BF** are only visible to debug mode. Implementations should raise illegal-instruction exceptions on machine-mode access to the latter set of registers.



Effective virtualization requires that as many instructions run natively as possible inside a virtualized environment, while any privileged accesses trap to the virtual machine monitor. (Goldberg, 1974) CSRs that are read-only at some lower privilege level are shadowed into separate CSR addresses if they are made read-write at a higher privilege level. This avoids trapping permitted lower-privilege accesses while still causing traps on illegal accesses. Currently, the counters are the only shadowed CSRs.

Table 99. Allocation of RISC-V CSR address ranges.

CSR Address			Hex	Use and Accessibility
[11:10]	[9:8]	[7:4]		
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000-0x0FF	Standard read/write
01	00	XXXX	0x400-0x4FF	Standard read/write
10	00	XXXX	0x800-0x8FF	Custom read/write
11	00	0XXX	0xC00-0xC7F	Standard read-only
11	00	10XX	0xC80-0xCBF	Standard read-only
11	00	11XX	0xCC0-0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100-0x1FF	Standard read/write
01	01	0XXX	0x500-0x57F	Standard read/write
01	01	10XX	0x580-0x5BF	Standard read/write
01	01	11XX	0x5C0-0x5FF	Custom read/write
10	01	0XXX	0x900-0x97F	Standard read/write
10	01	10XX	0x980-0x9BF	Standard read/write
10	01	11XX	0x9C0-0x9FF	Custom read/write
11	01	0XXX	0xD00-0xD7F	Standard read-only
11	01	10XX	0xD80-0xDBF	Standard read-only
11	01	11XX	0xDC0-0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200-0x2FF	Standard read/write
01	10	0XXX	0x600-0x67F	Standard read/write
01	10	10XX	0x680-0x6BF	Standard read/write
01	10	11XX	0x6C0-0x6FF	Custom read/write
10	10	0XXX	0xA00-0xA7F	Standard read/write
10	10	10XX	0xA80-0xABF	Standard read/write
10	10	11XX	0xAC0-0xAFF	Custom read/write
11	10	0XXX	0xE00-0xE7F	Standard read-only
11	10	10XX	0xE80-0xEBF	Standard read-only
11	10	11XX	0xEC0-0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300-0x3FF	Standard read/write
01	11	0XXX	0x700-0x77F	Standard read/write
01	11	100X	0x780-0x79F	Standard read/write
01	11	1010	0x7A0-0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0-0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0-0x7FF	Custom read/write
10	11	0XXX	0xB00-0xB7F	Standard read/write

10	11	10XX	0xB80-0xBBF	Standard read/write
10	11	11XX	0xBC0-0xBFF	Custom read/write
11	11	0XXX	0xF00-0xF7F	Standard read-only
11	11	10XX	0xF80-0xFBF	Standard read-only
11	11	11XX	0xFC0-0xFFF	Custom read-only

DRAFT

2.2. CSR Listing

Table 100-Table 103 list the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are standard unprivileged CSRs. The other registers are used by privileged code, as described in the following chapters. Note that not all registers are required on all implementations.

2.2.1. Currently allocated RISC-V unprivileged CSR addresses

Table 100. Currently allocated RISC-V unprivileged CSR addresses.

Number	Privilege	Name	Description
Unprivileged Floating-Point CSRs			
0x001	URW	fflags	Floating-Point Accrued Exceptions.
0x002	URW	frm	Floating-Point Dynamic Rounding Mode.
0x003	URW	fcsr	Floating-Point Control and Status Register (frm + fflags).
Unprivileged Vector CSRs			
0x008	URW	vstart	Vector start position.
0x009	URW	vxsat	Fixed-point accrued saturation flag.
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	Vector register length in bytes.
Unprivileged Zicfiss extension CSR			
0x011	URW	ssp	Shadow Stack Pointer.
Unprivileged Entropy Source Extension CSR			
0x015	URW	seed	Seed for cryptographic random bit generators.
Unprivileged Zcmt Extension CSR			
0x017	URW	jvt	Table jump base vector and control register.
Unprivileged Counter/Timers			
0xC00	URO	cycle	Cycle counter for RDCYCLE instruction.
0xC01	URO	time	Timer for RDTIME instruction.
0xC02	URO	instret	Instructions-retired counter for RDINSTRET instruction.
0xC03	URO	hpmcounter3	Performance-monitoring counter.
0xC04	URO	hpmcounter4	Performance-monitoring counter.
		:	
0xC1F	URO	hpmcounter31	Performance-monitoring counter.
0xC80	URO	cycleh	Upper 32 bits of cycle , RV32 only.
0xC81	URO	timeh	Upper 32 bits of time , RV32 only.
0xC82	URO	instreth	Upper 32 bits of instret , RV32 only.
0xC83	URO	hpmcounter3h	Upper 32 bits of hpmcounter3 , RV32 only.
0xC84	URO	hpmcounter4h	Upper 32 bits of hpmcounter4 , RV32 only.
		:	
0xC9F	URO	hpmcounter31h	Upper 32 bits of hpmcounter31 , RV32 only.

2.2.2. Currently allocated RISC-V supervisor-level CSR addresses

Table 101. Currently allocated RISC-V supervisor-level CSR addresses.

Number	Privilege	Name	Description
Supervisor Trap Setup			
0x100	SRW	sstatus	Supervisor status register.
0x104	SRW	sie	Supervisor interrupt-enable register.
0x105	SRW	stvec	Supervisor trap handler base address.
0x106	SRW	scounteren	Supervisor counter enable.
Supervisor Configuration			
0x10A	SRW	senvcfg	Supervisor environment configuration register.
Supervisor Counter Setup			
0x120	SRW	scountinhibit	Supervisor counter-inhibit register.
Supervisor Trap Handling			
0x140	SRW	sscratch	Supervisor scratch register.
0x141	SRW	sepc	Supervisor exception program counter.
0x142	SRW	scause	Supervisor trap cause.
0x143	SRW	stval	Supervisor trap value.
0x144	SRW	sip	Supervisor interrupt pending.
0xDA0	SRO	scountovf	Supervisor count overflow.
Supervisor Indirect			
0x150	SRW	siselect	Supervisor indirect register select.
0x151	SRW	sireg	Supervisor indirect register alias.
0x152	SRW	sireg2	Supervisor indirect register alias 2.
0x153	SRW	sireg3	Supervisor indirect register alias 3.
0x155	SRW	sireg4	Supervisor indirect register alias 4.
0x156	SRW	sireg5	Supervisor indirect register alias 5.
0x157	SRW	sireg6	Supervisor indirect register alias 6.
Supervisor Protection and Translation			
0x180	SRW	satp	Supervisor address translation and protection.
Supervisor Timer Compare			
0x14D	SRW	stimecmp	Supervisor timer compare.
0x15D	SRW	stimecmph	Upper 32 bits of stimecmp , RV32 only.
Debug/Trace Registers			
0x5A8	SRW	scontext	Supervisor-mode context register.
Supervisor Resource Management Configuration			
0x181	SRW	srmcfg	Supervisor Resource Management Configuration.
Supervisor State Enable Registers			
0x10C	SRW	sstateen0	Supervisor State Enable 0 Register.
0x10D	SRW	sstateen1	Supervisor State Enable 1 Register.
0x10E	SRW	sstateen2	Supervisor State Enable 2 Register.
0x10F	SRW	sstateen3	Supervisor State Enable 3 Register.
Supervisor Control Transfer Records Configuration			

Number	Privilege	Name	Description
0x14E	SRW	sctrctl	Supervisor Control Transfer Records Control Register.
0x14F	SRW	sctrstatus	Supervisor Control Transfer Records Status Register.
0x15F	SRW	sctrdepth	Supervisor Control Transfer Records Depth Register.

DRAFT

2.2.3. Currently allocated RISC-V hypervisor and VS CSR addresses

Table 102. Currently allocated RISC-V hypervisor and VS CSR addresses.

Number	Privilege	Name	Description
Hypervisor Trap Setup			
0x600	HRW	hstatus	Hypervisor status register.
0x602	HRW	hedeleg	Hypervisor exception delegation register.
0x603	HRW	hideleg	Hypervisor interrupt delegation register.
0x604	HRW	hie	Hypervisor interrupt-enable register.
0x606	HRW	hcounteren	Hypervisor counter enable.
0x607	HRW	hgeie	Hypervisor guest external interrupt-enable register.
0x612	HRW	hedelegh	Upper 32 bits of hedeleg , RV32 only.
Hypervisor Trap Handling			
0x643	HRW	htval	Hypervisor trap value.
0x644	HRW	hip	Hypervisor interrupt pending.
0x645	HRW	hvip	Hypervisor virtual interrupt pending.
0x64A	HRW	htinst	Hypervisor trap instruction (transformed).
0xE12	HRO	hgeip	Hypervisor guest external interrupt pending.
Hypervisor Configuration			
0x60A	HRW	henvcfg	Hypervisor environment configuration register.
0x61A	HRW	henvcfgh	Upper 32 bits of henvcfg , RV32 only.
Hypervisor Protection and Translation			
0x680	HRW	hgatp	Hypervisor guest address translation and protection.
Debug/Trace Registers			
0x6A8	HRW	hcontext	Hypervisor-mode context register.
Hypervisor Counter/Timer Virtualization Registers			
0x605	HRW	htimedelta	Delta for VS/VU-mode timer.
0x615	HRW	htimedeltah	Upper 32 bits of htimedelta , RV32 only.
Hypervisor State Enable Registers			
0x60C	HRW	hstateen0	Hypervisor State Enable 0 Register.
0x60D	HRW	hstateen1	Hypervisor State Enable 1 Register.
0x60E	HRW	hstateen2	Hypervisor State Enable 2 Register.
0x60F	HRW	hstateen3	Hypervisor State Enable 3 Register.
0x61C	HRW	hstateen0h	Upper 32 bits of Hypervisor State Enable 0 Register, RV32 only.
0x61D	HRW	hstateen1h	Upper 32 bits of Hypervisor State Enable 1 Register, RV32 only.
0x61E	HRW	hstateen2h	Upper 32 bits of Hypervisor State Enable 2 Register, RV32 only.
0x61F	HRW	hstateen3h	Upper 32 bits of Hypervisor State Enable 3 Register, RV32 only.
Virtual Supervisor Registers			

Number	Privilege	Name	Description
0x200	HRW	vsstatus	Virtual supervisor status register.
0x204	HRW	vsie	Virtual supervisor interrupt-enable register.
0x205	HRW	vstvec	Virtual supervisor trap handler base address.
0x240	HRW	vsscratch	Virtual supervisor scratch register.
0x241	HRW	vsepc	Virtual supervisor exception program counter.
0x242	HRW	vscause	Virtual supervisor trap cause.
0x243	HRW	vstval	Virtual supervisor trap value.
0x244	HRW	vsip	Virtual supervisor interrupt pending.
0x280	HRW	vsatp	Virtual supervisor address translation and protection.
Virtual Supervisor Indirect			
0x250	HRW	vsiselect	Virtual supervisor indirect register select.
0x251	HRW	vsireg	Virtual supervisor indirect register alias.
0x252	HRW	vsireg2	Virtual supervisor indirect register alias 2.
0x253	HRW	vsireg3	Virtual supervisor indirect register alias 3.
0x255	HRW	vsireg4	Virtual supervisor indirect register alias 4.
0x256	HRW	vsireg5	Virtual supervisor indirect register alias 5.
0x257	HRW	vsireg6	Virtual supervisor indirect register alias 6.
Virtual Supervisor Timer Compare			
0x24D	HRW	vstimecmp	Virtual supervisor timer compare.
0x25D	HRW	vstimecmph	Upper 32 bits of vstimecmp , RV32 only.
Virtual Supervisor Control Transfer Records Configuration			
0x24E	HRW	vsctrctl	Virtual Supervisor Control Transfer Records Control Register.

2.2.4. Currently allocated RISC-V machine-level CSR addresses

Table 103. Currently allocated RISC-V machine-level CSR addresses.

Number	Privilege	Name	Description
Machine Information Registers			
0xF11	MRO	mvendorid	Vendor ID.
0xF12	MRO	marchid	Architecture ID.
0xF13	MRO	mimpid	Implementation ID.
0xF14	MRO	mhartid	Hardware thread ID.
0xF15	MRO	mconfigptr	Pointer to configuration data structure.
Machine Trap Setup			
0x300	MRW	mstatus	Machine status register.
0x301	MRW	misa	ISA and extensions
0x302	MRW	medeleg	Machine exception delegation register.
0x303	MRW	mideleg	Machine interrupt delegation register.
0x304	MRW	mie	Machine interrupt-enable register.
0x305	MRW	mtvec	Machine trap-handler base address.
0x306	MRW	mcounteren	Machine counter enable.
0x310	MRW	mstatush	Additional machine status register, RV32 only.
0x312	MRW	medelegh	Upper 32 bits of medeleg , RV32 only.
Machine Trap Handling			
0x340	MRW	mscratch	Machine scratch register.
0x341	MRW	mepc	Machine exception program counter.
0x342	MRW	mcause	Machine trap cause.
0x343	MRW	mtval	Machine trap value.
0x344	MRW	mip	Machine interrupt pending.
0x34A	MRW	mtinst	Machine trap instruction (transformed).
0x34B	MRW	mtval2	Machine second trap value.
Machine Indirect			
0x350	MRW	miselect	Machine indirect register select.
0x351	MRW	mireg	Machine indirect register alias.
0x352	MRW	mireg2	Machine indirect register alias 2.
0x353	MRW	mireg3	Machine indirect register alias 3.
0x355	MRW	mireg4	Machine indirect register alias 4.
0x356	MRW	mireg5	Machine indirect register alias 5.
0x357	MRW	mireg6	Machine indirect register alias 6.
Machine Configuration			
0x30A	MRW	menvcfg	Machine environment configuration register.
0x31A	MRW	menvcfg	Upper 32 bits of menvcfg , RV32 only.
0x747	MRW	mseccfg	Machine security configuration register.
0x757	MRW	mseccfg	Upper 32 bits of mseccfg , RV32 only.
Machine Memory Protection			

Number	Privilege	Name	Description
0x3A0	MRW	pmpcfg0	Physical memory protection configuration.
0x3A1	MRW	pmpcfg1	Physical memory protection configuration, RV32 only.
0x3A2	MRW	pmpcfg2	Physical memory protection configuration.
0x3A3	MRW	pmpcfg3	Physical memory protection configuration, RV32 only.
		...	
0x3AE	MRW	pmpcfg14	Physical memory protection configuration.
0x3AF	MRW	pmpcfg15	Physical memory protection configuration, RV32 only.
0x3B0	MRW	pmpaddr0	Physical memory protection address register.
0x3B1	MRW	pmpaddr1	Physical memory protection address register.
		...	
0x3EF	MRW	pmpaddr63	Physical memory protection address register.
Machine State Enable Registers			
0x30C	MRW	mstateen0	Machine State Enable 0 Register.
0x30D	MRW	mstateen1	Machine State Enable 1 Register.
0x30E	MRW	mstateen2	Machine State Enable 2 Register.
0x30F	MRW	mstateen3	Machine State Enable 3 Register.
0x31C	MRW	mstateen0h	Upper 32 bits of Machine State Enable 0 Register, RV32 only.
0x31D	MRW	mstateen1h	Upper 32 bits of Machine State Enable 1 Register, RV32 only.
0x31E	MRW	mstateen2h	Upper 32 bits of Machine State Enable 2 Register, RV32 only.
0x31F	MRW	mstateen3h	Upper 32 bits of Machine State Enable 3 Register, RV32 only.
Machine Non-Maskable Interrupt Handling			
0x740	MRW	mnscratch	Resumable NMI scratch register.
0x741	MRW	mnepc	Resumable NMI program counter.
0x742	MRW	mncause	Resumable NMI cause.
0x744	MRW	mnstatus	Resumable NMI status.
Machine Counter/Timers			
0xB00	MRW	mcycle	Machine cycle counter.
0xB02	MRW	minstret	Machine instructions-retired counter.
0xB03	MRW	mhpmcounter3	Machine performance-monitoring counter.
0xB04	MRW	mhpmcounter4	Machine performance-monitoring counter.
		⋮	
0xB1F	MRW	mhpmcounter31	Machine performance-monitoring counter.
0xB80	MRW	mcycleh	Upper 32 bits of mcycle , RV32 only.
0xB82	MRW	minstreth	Upper 32 bits of minstret , RV32 only.
0xB83	MRW	mhpmcounter3h	Upper 32 bits of mhpmcounter3 , RV32 only.
0xB84	MRW	mhpmcounter4h	Upper 32 bits of mhpmcounter4 , RV32 only.
		⋮	
0xB9F	MRW	mhpmcounter31h	Upper 32 bits of mhpmcounter31 , RV32 only.
Machine Counter Setup			

Number	Privilege	Name	Description
0x320	MRW	mcountinhibit	Machine counter-inhibit register.
0x321	MRW	mcyclecfg	Machine cycle counter configuration register.
0x322	MRW	minstretcfg	Machine instret counter configuration register.
0x323	MRW	mhpmevent3	Machine performance-monitoring event selector.
0x324	MRW	mhpmevent4	Machine performance-monitoring event selector.
		⋮	
0x33F	MRW	mhpmevent31	Machine performance-monitoring event selector.
0x721	MRW	mcyclecfgh	Upper 32 bits of mcyclecfg , RV32 only.
0x722	MRW	minstretcfgh	Upper 32 bits of minstretcfg , RV32 only.
0x723	MRW	mhpmevent3h	Upper 32 bits of mhpmevent3 , RV32 only.
0x724	MRW	mhpmevent4h	Upper 32 bits of mhpmevent4 , RV32 only.
		⋮	
0x73F	MRW	mhpmevent31h	Upper 32 bits of mhpmevent31 , RV32 only.
Machine Control Transfer Records Configuration			
0x34E	MRW	mctrctl	Machine Control Transfer Records Control Register.
Debug/Trace Registers (shared with Debug Mode)			
0x7A0	MRW	tselect	Debug/Trace trigger register select.
0x7A1	MRW	tdata1	First Debug/Trace trigger data register.
0x7A2	MRW	tdata2	Second Debug/Trace trigger data register.
0x7A3	MRW	tdata3	Third Debug/Trace trigger data register.
0x7A4	MRW	tinfo	Trigger info register.
0x7A5	MRW	tcontrol	Trigger control register.
0x7A8	MRW	mcontext	Machine-mode context register.
Debug Mode Registers			
0x7B0	DRW	dcsr	Debug control and status register.
0x7B1	DRW	dpc	Debug program counter.
0x7B2	DRW	dscratch0	Debug scratch register 0.
0x7B3	DRW	dscratch1	Debug scratch register 1.

2.2.5. Currently allocated RISC-V indirect CSR (Smcsrind) mappings

Table 104. Currently allocated RISC-V indirect CSR (Smcsrind) mappings - M-mode

miselect	mireg	mireg2	mireg3	mireg4	mireg5	mireg6
0x30	iprio0	none	none	none	none	none
...	...	...	...	...	...	...
0x3F	iprio15	none	none	none	none	none
0x70	eidelivery	none	none	none	none	none
0x71	0	none	none	none	none	none
0x72	eithreshold	none	none	none	none	none
0x73	0	none	none	none	none	none
...	...	...	...	...	...	...
0x7F	0	none	none	none	none	none
0x80	eip0	none	none	none	none	none
...	...	...	...	...	...	...
0xBF	eip63	none	none	none	none	none
0xC0	eie0	none	none	none	none	none
...	...	...	...	...	...	...
0xFF	eie63	none	none	none	none	none

Table 105. Currently allocated RISC-V indirect CSR (Smcsrind/Scsrind) mappings - S-mode

siselect	sireg	sireg2	sireg3	sireg4	sireg5	sireg6
0x30	iprio0	none	none	none	none	none
...	...	...	...	...	...	...
0x3F	iprio15	none	none	none	none	none
0x40	cycle	cyclecfg	none	cycleh	cyclecfgh	none
0x41	none	none	none	none	none	none
0x42	instret	instretcfg	none	instreth	instretcfgh	none
0x43	hpmcounter3	hpmevent3	none	hpmcounter3h	hpmevent3h	none
...	...	...	...	...	...	...
0x5F	hpmcounter31	hpmevent31	none	hpmcounter31h	hpmevent31h	none
0x70	eidelivery	none	none	none	none	none
0x71	0	none	none	none	none	none
0x72	eithreshold	none	none	none	none	none
0x73	0	none	none	none	none	none
...	...	...	...	...	...	...
0x7F	0	none	none	none	none	none
0x80	eip0	none	none	none	none	none
...	...	...	...	...	...	...
0xBF	eip63	none	none	none	none	none

siselect	sireg	sireg2	sireg3	sireg4	sireg5	sireg6
0xC0	eie0	none	none	none	none	none
...	...	...	...	...	...	...
0xFF	eie63	none	none	none	none	none
0x200	ctrsource0	ctrtarget0	ctrdata0	0	0	0
...	...	...	...	...	...	...
0x2FF	ctrsource255	ctrtarget255	ctrdata255	0	0	0

Table 106. Currently allocated RISC-V indirect CSR (Smcsrind/Sscsrind) mappings - VS-mode

vsiselect	vsireg	vsireg2	vsireg3	vsireg4	vsireg5	vsireg6
0x30	iprio0	none	none	none	none	none
...	...	...	...	...	...	...
0x3F	iprio15	none	none	none	none	none
0x70	eidelivery	none	none	none	none	none
0x71	0	none	none	none	none	none
0x72	eithreshold	none	none	none	none	none
0x73	0	none	none	none	none	none
...	...	...	...	...	...	...
0x7F	0	none	none	none	none	none
0x80	eip0	none	none	none	none	none
...	...	...	...	...	...	...
0xBF	eip63	none	none	none	none	none
0xC0	eie0	none	none	none	none	none
...	...	...	...	...	...	...
0xFF	eie63	none	none	none	none	none
0x200	ctrsource0	ctrtarget0	ctrdata0	0	0	0
...	...	...	...	...	...	...
0x2FF	ctrsource255	ctrtarget255	ctrdata255	0	0	0

2.3. CSR Field Specifications

The following definitions and abbreviations are used in specifying the behavior of fields within the CSRs.

2.3.1. Reserved Writes Preserve Values, Reads Ignore Values (WPRI)

Some whole read/write fields are reserved for future use. Software should ignore the values read from these fields, and should preserve the values held in these fields when writing values to other fields of the same register. For forward compatibility, implementations that do not furnish these fields must make them read-only zero. These fields are labeled **WPRI** in the register descriptions.



To simplify the software model, any backward-compatible future definition of previously reserved fields within a CSR must cope with the possibility that a non-atomic read/modify/write sequence is used to update other fields in the CSR. Alternatively, the

original CSR definition must specify that subfields can only be updated atomically, which may require a two-instruction clear bit/set bit sequence in general that can be problematic if intermediate values are not legal.

2.3.2. Write/Read Only Legal Values (WLRL)

Some read/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value. These fields are labeled **WLRL** in the register descriptions.



Hardware implementations need only implement enough state bits to differentiate between the supported values, but must always return the complete specified bit-encoding of any supported value when read.

Implementations are permitted but not required to raise an illegal-instruction exception if an instruction attempts to write a non-supported value to a **WLRL** field. Implementations can return arbitrary bit patterns on the read of a **WLRL** field when the last write was of an illegal value, but the value returned should deterministically depend on the illegal written value and the value of the field prior to the write.

2.3.3. Write Any Values, Reads Legal Values (WARL)

Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained. These fields are labeled **WARL** in the register descriptions.

Implementations will not raise an exception on writes of unsupported values to a **WARL** field. Implementations can return any legal value on the read of a **WARL** field when the last write was of an illegal value, but the legal value returned should deterministically depend on the illegal written value and the architectural state of the hart.

2.4. CSR Field Modulation

If a write to one CSR changes the set of legal values allowed for a field of a second CSR, then unless specified otherwise, the second CSR's field immediately gets an **UNSPECIFIED** value from among its new legal values. This is true even if the field's value before the write remains legal after the write; the value of the field may be changed in consequence of the write to the controlling CSR.

*As a special case of this rule, the value written to one CSR may control whether a field of a second CSR is writable (with multiple legal values) or is read-only. When a write to the controlling CSR causes the second CSR's field to change from previously read-only to now writable, that field immediately gets an **UNSPECIFIED** but legal value, unless specified otherwise.*



*Some CSR fields are, when writable, defined as aliases of other CSR fields. Let x be such a CSR field, and let y be the CSR field it aliases when writable. If a write to a controlling CSR causes field x to change from previously read-only to now writable, the new value of x is not **UNSPECIFIED** but instead immediately reflects the existing value of its alias y , as required.*

A change to the value of a CSR for this reason is not a write to the affected CSR and thus does not trigger any side effects specified for that CSR.

2.5. Implicit Reads of CSRs

Implementations sometimes perform *implicit* reads of CSRs. (For example, all S-mode instruction fetches implicitly read the `satp` CSR.) Unless otherwise specified, the value returned by an implicit read of a CSR is the same value that would have been returned by an explicit read of the CSR, using a CSR-access instruction in a sufficient privilege mode.

2.6. CSR Width Modulation

If the width of a CSR is changed (for example, by changing `SXLEN` or `UXLEN`, as described in [Section 3.1.6.3](#)), the values of the *writable* fields and bits of the new-width CSR are, unless specified otherwise, determined from the previous-width CSR as though by this algorithm:

1. The value of the previous-width CSR is copied to a temporary register of the same width.
2. For the read-only bits of the previous-width CSR, the bits at the same positions in the temporary register are set to zeros.
3. The width of the temporary register is changed to the new width. If the new width W is narrower than the previous width, the least-significant W bits of the temporary register are retained and the more-significant bits are discarded. If the new width is wider than the previous width, the temporary register is zero-extended to the wider width.
4. Each writable field of the new-width CSR takes the value of the bits at the same positions in the temporary register.

Changing the width of a CSR is not a read or write of the CSR and thus does not trigger any side effects.

2.7. Explicit Accesses to CSRs Wider than XLEN

If a standard CSR is wider than `XLEN` bits, then an explicit read of the CSR returns the register's least-significant `XLEN` bits, and an explicit write to the CSR modifies only the register's least-significant `XLEN` bits, leaving the upper bits unchanged.

Some standard CSRs, such as the counter CSRs of extension `Zicntr`, are always 64 bits, even when `XLEN=32` (RV32). For each such 64-bit CSR (for example, counter `time`), a corresponding 32-bit *high-half* CSR is usually defined with the same name but with the letter 'h' appended at the end (`timeh`). The high-half CSR aliases bits 63:32 of its namesake 64-bit CSR, thus providing a way for RV32 software to read and modify the otherwise-unreachable 32 bits.

Standard high-half CSRs are accessible only when the base RISC-V instruction set is RV32 (`XLEN=32`). For RV64 (when `XLEN=64`), the addresses of all standard high-half CSRs are reserved, so an attempt to access a high-half CSR typically raises an illegal-instruction exception.

Chapter 3. Machine-Level ISA, Version 1.13

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V hart. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation.

3.1. Machine-Level CSRs

In addition to the machine-level CSRs described in this section, M-mode code can access all CSRs at lower privilege levels.

3.1.1. Machine ISA (`misa`) Register

The `misa` CSR is a **WARL** read-write register reporting the ISA supported by the hart. This register must be readable in any implementation, but a value of zero can be returned to indicate the `misa` register has not been implemented, requiring that CPU capabilities be determined through a separate non-standard mechanism.

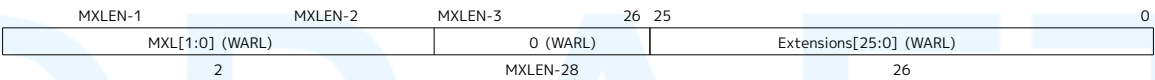


Figure 9. Machine ISA register (`misa`)

The `MXL` (Machine `XLEN`) field encodes the native base integer ISA width as shown in Table 107. The `MXL` field is read-only. If `misa` is nonzero, the `MXL` field indicates the effective `XLEN` in M-mode, a constant termed `MXLEN`. `XLEN` is never greater than `MXLEN`, but `XLEN` might be smaller than `MXLEN` in less-privileged modes.

Table 107. Encoding of `MXL` field in `misa`

MXL	XLEN
1	32
2	64
3	Reserved

The `misa` CSR is `MXLEN` bits wide.



The base width can be quickly ascertained using branches on the sign of the returned `misa` value, and possibly a shift left by one and a second branch on the sign. These checks can be written in assembly code without knowing the register width (`MXLEN`) of the hart. The base width is given by $MXLEN = 2^{MXL+4}$.

The base width can also be found if `misa` is zero, by placing the immediate 2 in a register, then shifting the register left by 31 bits. If zero, the hart is RV32, else it is RV64.

The Extensions field encodes the presence of the standard extensions, with a single bit per letter of the alphabet (bit 0 encodes presence of extension "A", bit 1 encodes presence of extension "B", through to bit 25 which encodes "Z"). The "I" bit will be set for the RV32I and RV64I base ISAs, and the "E" bit will be set for RV32E and RV64E. The Extensions field is a **WARL** field that can contain writable bits where the implementation allows the supported ISA to be modified. At reset, the Extensions field shall contain the maximal set of supported extensions, and "I" shall be selected over "E" if both are available.

When a standard extension is disabled by clearing its bit in **misa**, the instructions and CSRs defined or modified by the extension revert to their defined or reserved behaviors as if the extension is not implemented.



For a given RISC-V execution environment, an instruction, extension, or other feature of the RISC-V ISA is ordinarily judged to be implemented or not by the observable execution behavior in that environment. For example, the F extension is said to be implemented for an execution environment if and only if the instructions that the RISC-V Unprivileged ISA defines for F execute as specified.

*With this definition of implemented, disabling an extension by clearing its bit in **misa** results in the extension being considered not implemented in M-mode. For example, setting **misa.F**=0 results in the F extension being not implemented for M-mode, because the F extension's instructions will not act as the Unprivileged ISA requires but may instead raise an illegal-instruction exception.*

Defining the term implemented based strictly on the observable behavior might conflict with other common understandings of the same word. In particular, although common usage may allow for the combination "implemented but disabled," in this document it is considered a contradiction of terms, because disabled implies execution will not behave as required for the feature to be considered implemented. In the same vein, "implemented and enabled" is redundant here; "implemented" suffices.

All bits that are reserved for future use must return zero when read.

Table 108. Encoding of Extensions field in **misa**.

Bit	Character	Description
0	A	Atomic extension
1	B	B extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E/64E base ISA
5	F	Single-precision floating-point extension
6	G	<i>Reserved</i>
7	H	Hypervisor extension
8	I	RV32I/64I base ISA
9	J	<i>Reserved</i>
10	K	<i>Reserved</i>
11	L	<i>Reserved</i>
12	M	Integer Multiply/Divide extension
13	N	<i>Tentatively reserved for User-Level Interrupts extension</i>
14	O	<i>Reserved</i>
15	P	<i>Tentatively reserved for Packed-SIMD extension</i>
16	Q	Quad-precision floating-point extension
17	R	<i>Reserved</i>
18	S	Supervisor mode implemented
19	T	<i>Reserved</i>
20	U	User mode implemented
21	V	Vector extension
22	W	<i>Reserved</i>
23	X	Non-standard extensions implemented
24	Y	<i>Reserved</i>
25	Z	<i>Reserved</i>

The "X" bit will be set if there are any non-standard extensions.

When the "B" bit is 1, the implementation supports the instructions provided by the Zba, Zbb, and Zbs extensions. When the "B" bit is 0, it indicates that the implementation might not support one or more of the Zba, Zbb, or Zbs extensions.

When the "M" bit is 1, the implementation supports all multiply and division instructions defined by the M extension. When the "M" bit is 0, it indicates that the implementation might not support those instructions. However if the Zmmul extension is supported then the multiply instructions it specifies are supported irrespective of the value of the "M" bit.

When the "S" bit is 1, the implementation supports supervisor mode. When the "S" bit is 0, the implementation might not support supervisor mode.

When the "U" bit is 1, the implementation supports user mode. When the "U" bit is 0, the implementation might not support user mode.



The **mis** CSR exposes a rudimentary catalog of CPU features to machine-mode code. More extensive information can be obtained in machine mode by probing other machine registers, and examining other ROM storage in the system as part of the boot process.

We require that lower privilege levels execute environment calls instead of reading CPU registers to determine features available at each privilege level. This enables virtualization layers to alter the ISA observed at any level, and supports a much richer command interface

without burdening hardware designs.

The "E" bit is read-only. Unless **mis**a is all read-only zero, the "E" bit always reads as the complement of the "I" bit. If an execution environment supports both RV32E and RV32I, software can select RV32E by clearing the "I" bit.

If an ISA feature x depends on an ISA feature y , then attempting to enable feature x but disable feature y results in both features being disabled. For example, setting "F"=0 and "D"=1 results in both "F" and "D" being cleared. Similarly, setting "U"=0 and "S"=1 results in both "U" and "S" being cleared.

An implementation may impose additional constraints on the collective setting of two or more **mis**a fields, in which case they function collectively as a single **WARL** field. An attempt to write an unsupported combination causes those bits to be set to some supported combination.

Writing **mis**a may increase IALIGN, e.g., by disabling the "C" extension. If an instruction that would write **mis**a increases IALIGN, and the subsequent instruction's address is not IALIGN-bit aligned, the write to **mis**a is suppressed, leaving **mis**a unchanged.

When software enables an extension that was previously disabled, then all state uniquely associated with that extension is UNSPECIFIED, unless otherwise specified by that extension.



*Although one of the bits 25—0 in **mis**a being set to 1 implies that the corresponding feature is implemented, the inverse is not necessarily true: one of these bits being clear does not necessarily imply that the corresponding feature is not implemented. This follows from the fact that, when a feature is not implemented, the corresponding opcodes and CSRs become reserved, not necessarily illegal.*

3.1.2. Machine Vendor ID (**mvendorid**) Register

The **mvendorid** CSR is a 32-bit read-only register providing the JEDEC manufacturer ID of the provider of the core. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented or that this is a non-commercial implementation.

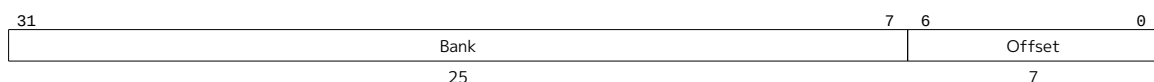


Figure 10. Vendor ID register (**mvendorid**)

JEDEC manufacturer IDs are ordinarily encoded as a sequence of one-byte continuation codes **0x7f**, terminated by a one-byte ID not equal to **0x7f**, with an odd parity bit in the most-significant bit of each byte. **mvendorid** encodes the number of one-byte continuation codes in the Bank field, and encodes the final byte in the Offset field, discarding the parity bit. For example, the JEDEC manufacturer ID **0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x7f 0x8a** (twelve continuation codes followed by **0x8a**) would be encoded in the **mvendorid** CSR as **0x60a**.



*In JEDEC's parlance, the bank number is one greater than the number of continuation codes; hence, the **mvendorid** Bank field encodes a value that is one less than the JEDEC bank number.*

Previously the vendor ID was to be a number allocated by RISC-V International, but this duplicates the work of JEDEC in maintaining a manufacturer ID standard. At time of writing, registering a manufacturer ID with JEDEC has a one-time cost of \$500.

3.1.3. Machine Architecture ID (**marchid**) Register

The **marchid** CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of **mvendorid** and **marchid** should uniquely identify the type of hart microarchitecture that is implemented.



Figure 11. Machine Architecture ID (**marchid**) register

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.

The intent is for the architecture ID to represent the microarchitecture associated with the project around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs are administered by RISC-V International and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

*The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The **misa** register also helps distinguish different variants of a design.*

3.1.4. Machine Implementation ID (**mimpid**) Register

The **mimpid** CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

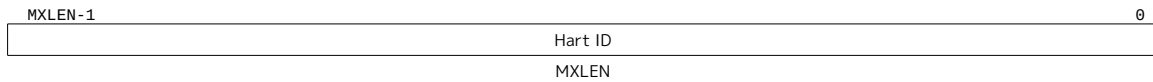


Figure 12. Machine Implementation ID (**mimpid**) register

The format of this field is left to the provider of the architecture source code, but will often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value can be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

3.1.5. Hart ID (**hartid**) Register

The **hartid** CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

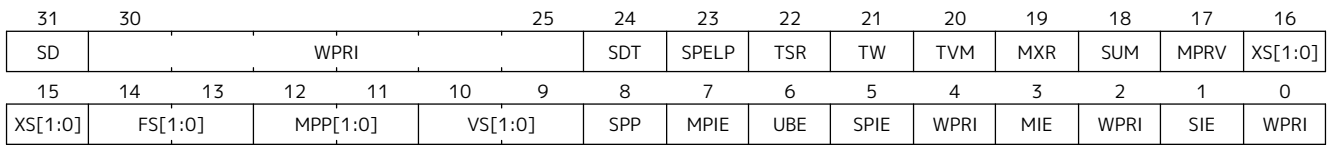
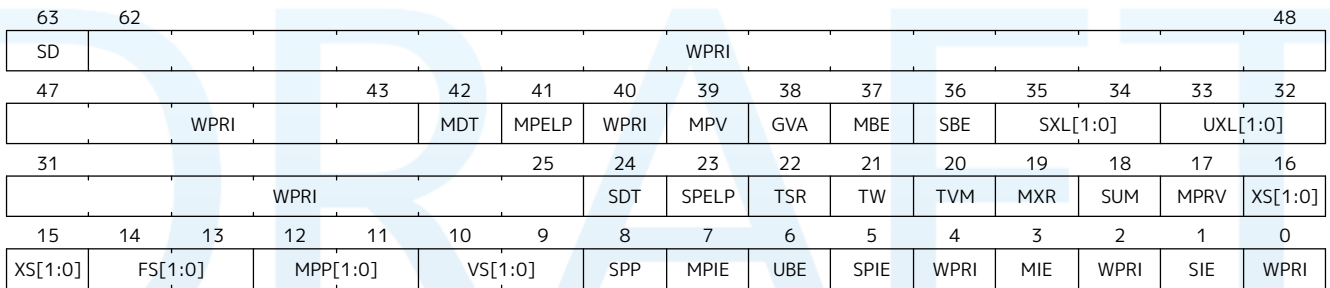
Figure 13. Hart ID (**mhartid**) register

In certain cases, we must ensure exactly one hart runs some code (e.g., at reset), and so require one hart to have a known hart ID of zero.

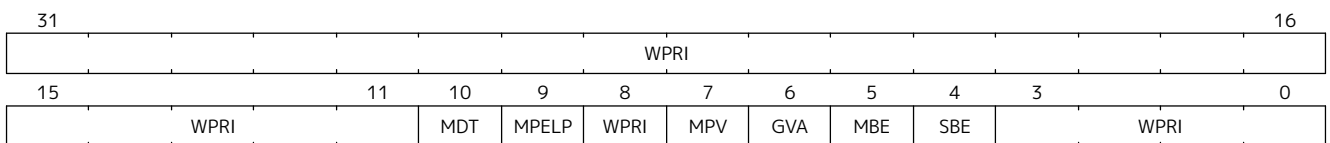
For efficiency, system implementers should aim to reduce the magnitude of the largest hart ID used in a system.

3.1.6. Machine Status (**mstatus** and **mstatush**) Registers

The **mstatus** register is an MXLEN-bit read/write register formatted as shown in Figure 14 for RV32 and Figure 15 for RV64. The **mstatus** register keeps track of and controls the hart's current operating state. A restricted view of **mstatus** appears as the **sstatus** register in the S-level ISA.

Figure 14. Machine-mode status (**mstatus**) register for RV32Figure 15. Machine-mode status (**mstatus**) register for RV64

For RV32 only, **mstatush** is a 32-bit read/write register formatted as shown in Figure 16. Bits 30:4 of **mstatush** generally contain the same fields found in bits 62:36 of **mstatus** for RV64. Fields SD, SXL, and UXL do not exist in **mstatush**.

Figure 16. Additional machine-mode status (**mstatush**) register for RV32.

3.1.6.1. Privilege and Global Interrupt-Enable Stack in **mstatus** register

Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively. These bits are primarily used to guarantee atomicity with respect to interrupt handlers in the current privilege mode.



The global *xIE* bits are located in the low-order bits of **mstatus**, allowing them to be atomically set or cleared with a single CSR instruction.

When a hart is executing in privilege mode *x*, interrupts are globally enabled when *xIE*=1 and globally disabled when *xIE*=0. Interrupts for lower-privilege modes, *w*<*x*, are always globally disabled regardless of the setting of any global *wIE* bit for the lower-privilege mode. Interrupts for higher-privilege modes, *y*>*x*,

are always globally enabled regardless of the setting of the global yIE bit for the higher-privilege mode. Higher-privilege-level code can use separate per-interrupt enable bits to disable selected higher-privilege-mode interrupts before ceding control to a lower-privilege mode. If supervisor mode is not implemented, then SIE and $SPIE$ are read-only 0.



A higher-privilege mode y could disable all of its interrupts before ceding control to a lower-privilege mode but this would be unusual as it would leave only a synchronous trap, non-maskable interrupt, or reset as means to regain control of the hart.

To support nested traps, each privilege mode x that can respond to interrupts has a two-level stack of interrupt-enable bits and privilege modes. $xPIE$ holds the value of the interrupt-enable bit active prior to the trap, and xPP holds the previous privilege mode. The xPP fields can only hold privilege modes up to x , so MPP is two bits wide and SPP is one bit wide. When a trap is taken from privilege mode y into privilege mode x , $xPIE$ is set to the value of xIE ; xIE is set to 0; and xPP is set to y .



For lower privilege modes, any trap (synchronous or asynchronous) is usually taken at a higher privilege mode with interrupts disabled upon entry. The higher-level trap handler will either service the trap and return using the stacked information, or, if not returning immediately to the interrupted context, will save the privilege stack before re-enabling interrupts, so only one entry per stack is required.

An $MRET$ or $SRET$ instruction is used to return from a trap in M-mode or S-mode respectively. When executing an $xRET$ instruction, supposing xPP holds the value y , xIE is set to $xPIE$; the privilege mode is changed to y ; $xPIE$ is set to 1; and xPP is set to the least-privileged supported mode (U if U-mode is implemented, else M). If $y \neq M$, $xRET$ also sets $MPRV=0$.



Setting xPP to the least-privileged supported mode on an $xRET$ helps identify software bugs in the management of the two-level privilege-mode stack.



Trap handlers must be designed to neither enable interrupts nor cause exceptions during the phase of handling where the trap handler preserves the critical state information required to handle and resume from the trap. An exception or interrupt in this critical phase of trap handling may lead to a trap that can overwrite such critical state. This could result in the loss of data needed to recover from the initial trap. Further, if an exception occurs in the code path needed to handle traps, then such a situation may lead to an infinite loop of traps. To prevent this, trap handlers must be meticulously designed to identify and safely manage exceptions within their operational flow.

xPP fields are **WARL** fields that can hold only privilege mode x and any implemented privilege mode lower than x . If privilege mode x is not implemented, then xPP must be read-only 0.



M-mode software can determine whether a privilege mode is implemented by writing that mode to MPP then reading it back.

If the machine provides only U and M modes, then only a single hardware storage bit is required to represent either 00 or 11 in MPP .

3.1.6.2. Double Trap Control in `mstatus` Register

A double trap typically arises during a sensitive phase in trap handling operations — when an exception or interrupt occurs while the trap handler (the component responsible for managing these events) is in a non-reentrant state. This non-reentrancy usually occurs in the early phase of trap handling, wherein the trap handler has not yet preserved the necessary state to handle and resume from the trap. The occurrence of a trap during this phase can lead to an overwrite of critical state information, resulting in the loss of data needed to recover from the initial trap. The trap that caused this critical error condition is henceforth

called the *unexpected trap*. Trap handlers are designed to neither enable interrupts nor cause exceptions during this phase of handling. However, managing Hardware-Error exceptions, which may occur unpredictably, presents significant challenges in trap handler implementation due to the potential risk of a double trap.

The M-mode-disable-trap (**MDT**) bit is a WARL field introduced by the Smdbltrp extension. Upon reset, the **MDT** field is set to 1. When the **MDT** bit is set to 1 by an explicit CSR write, the **MIE** (Machine Interrupt Enable) bit is cleared to 0. For RV64, this clearing occurs regardless of the value written, if any, to the **MIE** bit by the same write. The **MIE** bit can only be set to 1 by an explicit CSR write if the **MDT** bit is already 0 or, for RV64, is being set to 0 by the same write (For RV32, the **MDT** bit is in **mstatush** and the **MIE** bit in **mstatus** register).

When a trap is to be taken into M-mode, if the **MDT** bit is currently 0, it is then set to 1, and the trap is delivered as expected. However, if **MDT** is already set to 1, then this is an *unexpected trap*. When the Smrnmi extension is implemented, a trap caused by an RNMI is not considered an *unexpected trap* irrespective of the state of the **MDT** bit. A trap caused by an RNMI does not set the **MDT** bit. However, a trap that occurs when executing in M-mode with **mstatus.NMIE** set to 0 is an *unexpected trap*.

In the event of a *unexpected trap*, the handling is as follows:

- When the Smrnmi extension is implemented and **mstatus.NMIE** is 1, the hart traps to the RNMI handler. To deliver this trap, the **mnepc** and **mncause** registers are written with the values that the *unexpected trap* would have written to the **mepc** and **mcause** registers respectively. The privilege mode information fields in the **mstatus** register are written to indicate M-mode and its **NMIE** field is set to 0.



*The consequence of this specification is that on occurrence of double trap the RNMI handler is not provided with information that a trap reports in the **mtval** and the **mtval2** registers. This information, if needed, can be obtained by the RNMI handler by decoding the instruction at the address in **mnepc** and examining its source register contents.*

- When the Smrnmi extension is not implemented, or if the Smrnmi extension is implemented and **mstatus.NMIE** is 0, the hart enters a critical-error state without updating any architectural state, including the **pc**. This state involves ceasing execution, disabling all interrupts (including NMIs), and asserting a **critical-error** signal to the platform. Whether performance counters and timers are updated in the critical-error state is UNSPECIFIED.



*The actions performed by the platform when a hart asserts a **critical-error** signal are platform-specific. The range of possible actions include restarting the affected hart or restarting the entire platform, among others.*

The **MRET** and **SRET** instructions, when executed in M-mode, set the **MDT** bit to 0. If the new privilege mode is U, VS, or VU, then **sstatus.SDT** is also set to 0. Additionally, if it is VU, then **vsstatus.SDT** is also set to 0.

The **MNRET** instruction, provided by the Smrnmi extension, sets the **MDT** bit to 0 if the new privilege mode is not M. If it is U, VS, or VU, then **sstatus.SDT** is also set to 0. Additionally, if it is VU, then **vsstatus.SDT** is also set to 0.

3.1.6.3. Base ISA Control in **mstatus** Register

For RV64 harts, the **SXL** and **UXL** fields are **WARL** fields that control the value of **XLEN** for S-mode and U-mode, respectively. The encoding of these fields is the same as the **MXL** field of **misal**, shown in [Table 107](#). The effective **XLEN** in S-mode and U-mode are termed **SXLEN** and **UXLEN**, respectively.

When **MXLEN**=32, the **SXL** and **UXL** fields do not exist, and **SXLEN**=32 and **UXLEN**=32.

When **MXLEN**=64, if S-mode is not supported, then **SXL** is read-only zero. Otherwise, it is a **WARL** field

that encodes the current value of SXLEN. In particular, an implementation may make SXL be a read-only field whose value always ensures that SXLEN=MXLEN.

When MXLEN=64, if U-mode is not supported, then UXL is read-only zero. Otherwise, it is a **WARL** field that encodes the current value of UXLEN. In particular, an implementation may make UXL be a read-only field whose value always ensures that UXLEN=MXLEN or UXLEN=SXLEN.

If S-mode is implemented, the set of legal values that the UXL field may assume excludes those that would cause UXLEN to be greater than SXLEN.

Whenever XLEN in any mode is set to a value less than the widest supported XLEN, all operations must ignore source operand register bits above the configured XLEN, and must sign-extend results to fill the entire widest supported XLEN in the destination register. Similarly, **pc** bits above XLEN are ignored, and when the **pc** is written, it is sign-extended to fill the widest supported XLEN.

We require that operations always fill the entire underlying hardware registers with defined values to avoid implementation-defined behavior.



To reduce hardware complexity, the architecture imposes no checks that lower-privilege modes have XLEN settings less than or equal to the next-higher privilege mode. In practice, such settings would almost always be a software bug, but machine operation is well-defined even in this case.

Some HINT instructions are encoded as integer computational instructions that overwrite their destination register with its current value, e.g., **c.addi x8, 0**. When such a HINT is executed with XLEN < MXLEN and bits MXLEN..XLEN of the destination register not all equal to bit XLEN-1, it is implementation-defined whether bits MXLEN..XLEN of the destination register are unchanged or are overwritten with copies of bit XLEN-1.



This definition allows implementations to elide register write-back for some HINTs, while allowing them to execute other HINTs in the same manner as other integer computational instructions. The implementation choice is observable only by privilege modes with an XLEN setting greater than the current XLEN; it is invisible to the current privilege mode.

3.1.6.4. Memory Privilege in **mstatus** Register

The MPRV (Modify PRiVilege) bit modifies the *effective privilege mode*, i.e., the privilege level at which explicit memory accesses execute. When MPRV=0, explicit memory accesses behave as normal, using the translation and protection mechanisms of the current privilege mode. When MPRV=1, load and store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP. Instruction address-translation and protection are unaffected by the setting of MPRV. MPRV is read-only 0 if U-mode is not supported.

An MRET or SRET instruction that changes the privilege mode to a mode less privileged than M also sets MPRV=0.

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in [Figure 60](#)) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect. MXR is read-only 0 if S-mode is not supported.



The MPRV and MXR mechanisms were conceived to improve the efficiency of M-mode routines that emulate missing hardware features, e.g., misaligned loads and stores. MPRV obviates the need to perform address translation in software. MXR allows instruction words to be loaded from pages marked execute-only.

The current privilege mode and the privilege mode specified by MPP might have different XLEN settings. When MPRV=1, load and store memory addresses are treated as though the current XLEN were set to MPP's XLEN, following the rules in [Section 3.1.6.3](#).

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in [Figure 60](#)) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect. Note that, while SUM is ordinarily ignored when not executing in S-mode, it is in effect when MPRV=1 and MPP=S. SUM is read-only 0 if S-mode is not supported or if `satp.MODE` is read-only 0.

The MXR and SUM mechanisms only affect the interpretation of permissions encoded in page-table entries. In particular, they have no impact on whether access-fault exceptions are raised due to PMAs or PMP.

3.1.6.5. Endianness Control in `mstatus` and `mstatush` Registers

The MBE, SBE, and UBE bits in `mstatus` and `mstatush` are WARL fields that control the endianness of memory accesses other than instruction fetches. Instruction fetches are always little-endian.

MBE controls whether non-instruction-fetch memory accesses made from M-mode (assuming `mstatus.MPRV=0`) are little-endian (MBE=0) or big-endian (MBE=1).

If S-mode is not supported, SBE is read-only 0. Otherwise, SBE controls whether explicit load and store memory accesses made from S-mode are little-endian (SBE=0) or big-endian (SBE=1).

If U-mode is not supported, UBE is read-only 0. Otherwise, UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1).

For *implicit* accesses to supervisor-level memory management data structures, such as page tables, endianness is always controlled by SBE. Since changing SBE alters the implementation's interpretation of these data structures, if any such data structures remain in use across a change to SBE, M-mode software must follow such a change to SBE by executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0`.



Only in contrived scenarios will a given memory-management data structure be interpreted as both little-endian and big-endian. In practice, SBE will only be changed at runtime on world switches, in which case neither the old nor new memory-management data structure will be reinterpreted in a different endianness. In this case, no additional `SFENCE.VMA` is necessary, beyond what would ordinarily be required for a world switch.

If S-mode is supported, an implementation may make SBE be a read-only copy of MBE. If U-mode is supported, an implementation may make UBE be a read-only copy of either MBE or SBE.

An implementation supports only little-endian memory accesses if fields MBE, SBE, and UBE are all read-only 0. An implementation supports only big-endian memory accesses (aside from instruction fetches) if MBE is read-only 1 and SBE and UBE are each read-only 1 when S-mode and U-mode are supported.



[Volume I, Section 1.4](#) defines a hart's address space as a circular sequence of 2^{XLEN} bytes at consecutive addresses. The correspondence between addresses and byte locations is fixed and not affected by any endianness mode. Rather, the applicable endianness mode determines the order of mapping between memory bytes and a multibyte quantity (halfword, word, etc.).

Standard RISC-V ABIs are expected to be purely little-endian-only or big-endian-only, with no accommodation for mixing endianness. Nevertheless, endianness control has been defined so as to permit, for instance, an OS of one endianness to execute user-mode programs of the opposite endianness. Consideration has been given also to the possibility of non-standard usages whereby software flips the endianness of memory accesses as needed.

RISC-V instructions are uniformly little-endian to decouple instruction encoding from the current endianness settings, for the benefit of both hardware and software. Otherwise, for instance, a RISC-V assembler or disassembler would always need to know the intended active endianness, despite that 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. In big-endian mode, such software must account for the fact that explicit loads and stores have endianness opposite that of instructions, for example by swapping byte order after loads and before stores.

3.1.6.6. Virtualization Support in `mstatus` Register

The TVM (Trap Virtual Memory) bit is a **WARL** field that supports intercepting supervisor virtual-memory management operations. When TVM=1, attempts to read or write the `satp` CSR or execute an `SFENCE.VMA` or `SINVAL.VMA` instruction while executing in S-mode will raise an illegal-instruction exception. When TVM=0, these operations are permitted in S-mode. TVM is read-only 0 when S-mode is not supported.

The TVM mechanism improves virtualization efficiency by permitting guest operating systems to execute in S-mode, rather than classically virtualizing them in U-mode. This approach obviates the need to trap accesses to most S-mode CSRs.

Trapping `satp` accesses and the `SFENCE.VMA` and `SINVAL.VMA` instructions provides the hooks necessary to lazily populate shadow page tables.

The TW (Timeout Wait) bit is a **WARL** field that supports intercepting the WFI instruction (see [Section 3.3.3](#)). When TW=0, the WFI instruction may execute in modes less privileged than M when not prevented for some other reason. When TW=1, then if WFI is executed in any less-privileged mode, and it does not complete within an implementation-specific, bounded time limit, the WFI instruction causes an illegal-instruction exception. An implementation may have WFI always raise an illegal-instruction exception in modes less privileged than M when TW=1, even if there are pending globally-disabled interrupts when the instruction is executed. TW is read-only 0 when there are no modes less privileged than M.

Trapping the WFI instruction can trigger a world switch to another guest OS, rather than wastefully idling in the current guest.

When S-mode is implemented, then executing WFI in U-mode causes an illegal-instruction exception, regardless of the value of the TW bit, unless the instruction completes within an implementation-specific, bounded time limit.

The TSR (Trap SRET) bit is a **WARL** field that supports intercepting the supervisor exception return instruction, SRET. When TSR=1, attempts to execute SRET while executing in S-mode will raise an illegal-instruction exception. When TSR=0, this operation is permitted in S-mode. TSR is read-only 0 when S-

mode is not supported.



Trapping SRET is necessary to emulate the hypervisor extension (see [Chapter 5](#)) on implementations that do not provide it.

3.1.6.7. Extension Context Status in mstatus Register

Supporting substantial extensions is one of the primary goals of RISC-V, and hence we define a standard interface to allow unchanged privileged-mode code, particularly a supervisor-level OS, to support arbitrary user-mode state extensions.



To date, the V extension is the only standard extension that defines additional state beyond the floating-point CSR and data registers.

The FS[1:0] and VS[1:0] WARL fields and the XS[1:0] read-only field are used to reduce the cost of context save and restore by setting and tracking the current state of the floating-point unit and any other user-mode extensions respectively. The FS field encodes the status of the floating-point unit state, including the floating-point registers **f0–f31** and the CSRs **fcsr**, **frm**, and **fflags**. The VS field encodes the status of the vector extension state, including the vector registers **v0–v31** and the CSRs **vcsr**, **vxrm**, **vxsat**, **vstart**, **vl**, **vtype**, and **vlenb**. The XS field encodes the status of additional user-mode extensions and associated state. These fields can be checked by a context switch routine to quickly determine whether a state save or restore is required. If a save or restore is required, additional instructions and CSRs are typically required to effect and optimize the process.



The design anticipates that most context switches will not need to save/restore state in either or both of the floating-point unit or other extensions, so provides a fast check via the SD bit.

The FS, VS, and XS fields use the same status encoding as shown in [Table 109](#), with the four possible status values being Off, Initial, Clean, and Dirty.

Table 109. Encoding of FS[1:0], VS[1:0], and XS[1:0] status fields

Status	FS and VS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

If the F extension is implemented, the FS field shall not be read-only zero.

If neither the F extension nor S-mode is implemented, then FS is read-only zero. If S-mode is implemented but the F extension is not, FS may optionally be read-only zero.



Implementations with S-mode but without the F extension are permitted, but not required, to make the FS field be read-only zero. Some such implementations will choose not to have the FS field be read-only zero, so as to enable emulation of the F extension for both S-mode and U-mode via invisible traps into M-mode.

If the **v** registers are implemented, the VS field shall not be read-only zero.

If neither the **v** registers nor S-mode is implemented, then VS is read-only zero. If S-mode is implemented but the **v** registers are not, VS may optionally be read-only zero.

In harts without additional user extensions requiring new state, the XS field is read-only zero. Every additional extension with state provides a CSR field that encodes the equivalent of the XS states. The XS

field represents a summary of all extensions' status as shown in [Table 109](#).



The XS field effectively reports the maximum status value across all user-extension status fields, though individual extensions can use a different encoding than XS.

The SD bit is a read-only bit that summarizes whether either the FS, VS, or XS fields signal the presence of some dirty state that will require saving extended user context to memory. If FS, XS, and VS are all read-only zero, then SD is also always zero.

When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an illegal-instruction exception. When the status is Initial, the corresponding state should have an initial constant value. When the status is Clean, the corresponding state is potentially different from the initial value, but matches the last value stored on a context swap. When the status is Dirty, the corresponding state has potentially been modified since the last context save.

During a context save, the responsible privileged code need only write out the corresponding state if its status is Dirty, and can then reset the extension's status to Clean. During a context restore, the context need only be loaded from memory if the status is Clean (it should never be Dirty at restore). If the status is Initial, the context must be set to an initial constant value on context restore to avoid a security hole, but this can be done without accessing memory. For example, the floating-point registers can all be initialized to the immediate value 0.

The FS and XS fields are read by the privileged code before saving the context. The FS field is set directly by privileged code when resuming a user context, while the XS field is set indirectly by writing to the status register of the individual extensions. The status fields will also be updated during execution of instructions, regardless of privilege mode.

Extensions to the user-mode ISA often include additional user-mode state, and this state can be considerably larger than the base integer registers. The extensions might only be used for some applications, or might only be needed for short phases within a single application. To improve performance, the user-mode extension can define additional instructions to allow user-mode software to return the unit to an initial state or even to turn off the unit.

For example, a coprocessor might require to be configured before use and can be "unconfigured" after use. The unconfigured state would be represented as the Initial state for context save. If the same application remains running between the unconfigure and the next configure (which would set status to Dirty), there is no need to actually reinitialize the state at the unconfigure instruction, as all state is local to the user process, i.e., the Initial state may only cause the coprocessor state to be initialized to a constant value at context restore, not at every unconfigure.

Executing a user-mode instruction to disable a unit and place it into the Off state will cause an illegal-instruction exception to be raised if any subsequent instruction tries to use the unit before it is turned back on. A user-mode instruction to turn a unit on must also ensure the unit's state is properly initialized, as the unit might have been used by another context meantime.

Changing the setting of FS has no effect on the contents of the floating-point register state. In particular, setting FS=Off does not destroy the state, nor does setting FS=Initial clear the contents. Similarly, the setting of VS has no effect on the contents of the vector register state. Other extensions, however, might not preserve state when set to Off.

Implementations may choose to track the dirtiness of the floating-point register state imprecisely by reporting the state to be dirty even when it has not been modified. On some implementations, some instructions that do not mutate the floating-point state may cause the state to transition from Initial or Clean to Dirty. On other implementations, dirtiness might not be tracked at all, in which case the valid FS states are Off and Dirty, and an attempt to set FS to Initial or Clean causes it to be set to Dirty.



This definition of FS does not disallow setting FS to Dirty as a result of errant speculation. Some platforms may choose to disallow speculatively writing FS to close a potential side channel.

If an instruction explicitly or implicitly writes a floating-point register or the **fcsr** but does not alter its contents, and FS=Initial or FS=Clean, it is implementation-defined whether FS transitions to Dirty.

Implementations may choose to track the dirtiness of the vector register state in an analogous imprecise fashion, including possibly setting VS to Dirty when software attempts to set VS=Initial or VS=Clean. When VS=Initial or VS=Clean, it is implementation-defined whether an instruction that writes a vector register or vector CSR but does not alter its contents causes VS to transition to Dirty.

Table 110 shows all the possible state transitions for the FS, VS, or XS status bits. Note that the standard floating-point and vector extensions do not support user-mode unconfigure or disable/enable instructions.

DRAFT

Table 110. FS, VS, and XS state transitions.

Current State Action	Off	Initial	Clean	Dirty
At context save in privileged code				
Save state? Next state	No Off	No Initial	No Clean	Yes Clean
At context restore in privileged code				
Restore state? Next state	No Off	Yes, to initial Initial	Yes, from memory Clean	N/A N/A
Execute instruction to read state				
Action? Next state	Exception Off	Execute Initial	Execute Clean	Execute Dirty
Execute instruction that possibly modifies state, including configuration				
Action? Next state	Exception Off	Execute Dirty	Execute Dirty	Execute Dirty
Execute instruction to unconfigure unit				
Action? Next state	Exception Off	Execute Initial	Execute Initial	Execute Initial
Execute instruction to disable unit				
Action? Next state	Execute Off	Execute Off	Execute Off	Execute Off
Execute instruction to enable unit				
Action? Next state	Execute Initial	Execute Initial	Execute Initial	Execute Initial

Standard privileged instructions to initialize, save, and restore extension state are provided to insulate privileged code from details of the added extension state by treating the state as an opaque object.



Many coprocessor extensions are only used in limited contexts that allows software to safely unconfigure or even disable units when done. This reduces the context-switch overhead of large stateful coprocessors.

We separate out floating-point state from other extension state, as when a floating-point unit is present the floating-point registers are part of the standard calling convention, and so user-mode software cannot know when it is safe to disable the floating-point unit.

The XS field provides a summary of all added extension state, but additional microarchitectural bits might be maintained in the extension to further reduce context save and restore overhead.

The SD bit is read-only and is set when either the FS, VS, or XS bits encode a Dirty state (i.e., **SD=(FS==0b11 OR XS==0b11 OR VS==0b11)**). This allows privileged code to quickly determine when no additional context save is required beyond the integer register set and **pc**.

The floating-point unit state is always initialized, saved, and restored using standard instructions (F, D, and/or Q), and privileged code must be aware of FLEN to determine the appropriate space to reserve for each **f** register.

Machine and Supervisor modes share a single copy of the FS, VS, and XS bits. Supervisor-level software normally uses the FS, VS, and XS bits directly to record the status with respect to the supervisor-level saved context. Machine-level software must be more conservative in saving and restoring the extension state in their corresponding version of the context.



In any reasonable use case, the number of context switches between user and supervisor level should far outweigh the number of context switches to other privilege levels. Note that coprocessors should not require their context to be saved and restored to service asynchronous interrupts, unless the interrupt results in a user-level context swap.

3.1.6.8. Previous Expected Landing Pad (ELP) State in **mstatus** Register

The Zicfilp extension adds the **SPELP** and **MPELP** fields that hold the previous ELP, and are updated as specified in [Section 6.9.1.2](#). The **xPELP** fields are encoded as follows:

- 0 - **NO_LP_EXPECTED** - no landing pad instruction expected.
- 1 - **LP_EXPECTED** - a landing pad instruction is expected.

3.1.7. Machine Trap-Vector Base-Address (**mtvec**) Register

The **mtvec** register is an MXLEN-bit **WARL** read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).

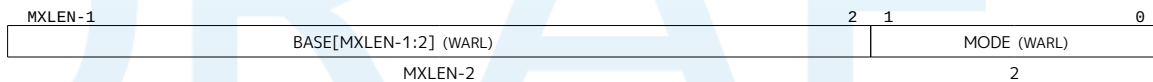


Figure 17. Encoding of **mtvec** MODE field.

The **mtvec** register must always be implemented, but can contain a read-only value. If **mtvec** 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 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field. Note that the CSR contains only bits XLEN-1 through 2 of the address BASE. When used as an address, the lower two bits are filled with zeroes to obtain an XLEN-bit address that is always aligned on a 4-byte boundary.



We allow for considerable flexibility in implementation of the trap vector base address. On the one hand, we do not wish to burden low-end implementations with a large number of state bits, but on the other hand, we wish to allow flexibility for larger systems.

Table 111. Encoding of **mtvec** MODE field.

Value	Name	Description
0	Direct	All traps set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	---	Reserved

The encoding of the MODE field is shown in [Table 111](#). When MODE=Direct, all traps into machine mode cause the **pc** to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the **pc** to be set to the address in the BASE field, whereas interrupts cause the **pc** to be set to the address in the BASE field plus four times the interrupt cause number. For example, a

machine-mode timer interrupt (see [Table 112](#)) causes the **pc** to be set to **BASE+0x1c**.

An implementation may have different alignment constraints for different modes. In particular, **MODE=Vectored** may have stricter alignment constraints than **MODE=Direct**.



Allowing coarser alignments in Vectored mode enables vectoring to be implemented without a hardware adder circuit.

Reset and NMI vector locations are given in a platform specification.

3.1.8. Machine Trap Delegation (**medeleg** and **mideleg**) Registers

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the **MRET** instruction ([Section 3.3.2](#)). To increase performance, implementations can provide individual read/write bits within **medeleg** and **mideleg** to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (**medeleg**) is a 64-bit read/write register. The machine interrupt delegation (**mideleg**) register is an **MXLEN**-bit read/write register.

In harts with S-mode, the **medeleg** and **mideleg** registers must exist, and setting a bit in **medeleg** or **mideleg** will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. In harts without S-mode, the **medeleg** and **mideleg** registers should not exist.



*In versions 1.9.1 and earlier, these registers existed but were hardwired to zero in M-mode only, or M/U without N harts. There is no reason to require they return zero in those cases, as the **mtval** register indicates whether they exist.*

When a trap is delegated to S-mode, the **scause** register is written with the trap cause; the **sepc** register is written with the virtual address of the instruction that took the trap; the **stval** register is written with an exception-specific datum; the **SPP** field of **mstatus** is written with the active privilege mode at the time of the trap; the **SPIE** field of **mstatus** is written with the value of the **SIE** field at the time of the trap; and the **SIE** field of **mstatus** is cleared. The **mcause**, **mepc**, and **mtval** registers and the **MPP** and **MPIE** fields of **mstatus** are not written.

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in **medeleg** or **mideleg** to see which bit positions hold a one.

An implementation shall not have any bits of **medeleg** be read-only one, i.e., any synchronous trap that can be delegated must support not being delegated. Similarly, an implementation shall not fix as read-only one any bits of **mideleg** corresponding to machine-level interrupts (but may do so for lower-level interrupts).



*Version 1.11 and earlier prohibited having any bits of **mideleg** be read-only one. Platform standards may always add such restrictions.*

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal-instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may be taken horizontally. Using the same example, if M-mode has delegated illegal-instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode.

Delegated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting **mideleg**[5], STIs will not be taken

when executing in M-mode. By contrast, if `mideleg`[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

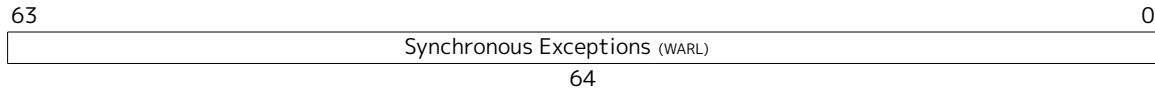


Figure 18. Machine Exception Delegation (`mideleg`) register.

`mideleg` has a bit position allocated for every synchronous exception shown in Table 112, with the index of the bit position equal to the value returned in the `mcause` register (i.e., setting bit 8 allows user-mode environment calls to be delegated to a lower-privilege trap handler).

When `XLEN`=32, `midelegh` is a 32-bit read/write register that aliases bits 63:32 of `mideleg`. The `midelegh` register does not exist when `XLEN`=64.



Figure 19. Machine Interrupt Delegation (`mideleg`) Register.

`mideleg` holds trap delegation bits for individual interrupts, with the layout of bits matching those in the `mip` register (i.e., STIP interrupt delegation control is located in bit 5).

For exceptions that cannot occur in less privileged modes, the corresponding `mideleg` bits should be read-only zero. In particular, `mideleg`[11] is read-only zero.

The `mideleg`[16] is read-only zero as double trap is not delegatable.

3.1.9. Machine Interrupt (`mip` and `mie`) Registers

The `mip` register is an `MXLEN`-bit read/write register containing information on pending interrupts, while `mie` is the corresponding `MXLEN`-bit read/write register containing interrupt enable bits. Interrupt cause number i (as reported in CSR `mcause`, Section 3.1.15) corresponds with bit i in both `mip` and `mie`. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.



Interrupts designated for platform use may be designated for custom use at the platform's discretion.

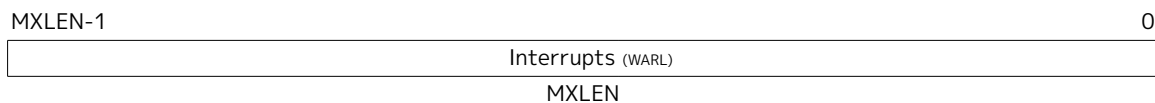


Figure 20. Machine Interrupt-Pending (`mip`) register.



Figure 21. Machine Interrupt-Enable (`mie`) register

An interrupt i will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true: (a) either the current privilege mode is M and the MIE bit in the `mstatus` register is set, or the current privilege mode has less privilege than M-mode; (b) bit i is set in both `mip` and `mie`; and (c) if register `mideleg` exists, bit i is not set in `mideleg`.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in `mip`, and must also be evaluated immediately following

the execution of an `xRET` instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including `mip`, `mie`, `mstatus`, and `mideleg`).

Interrupts to M-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register `mip` may be writable or may be read-only. When bit i in `mip` is writable, a pending interrupt i can be cleared by writing 0 to this bit. If interrupt i can become pending but bit i in `mip` is read-only, the implementation must provide some other mechanism for clearing the pending interrupt.

A bit in `mie` must be writable if the corresponding interrupt can ever become pending. Bits of `mie` that are not writable must be read-only zero.

The standard portions (bits 15:0) of the `mip` and `mie` registers are formatted as shown in Figure 22 and Figure 23 respectively.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIP	0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 22. Standard portion (bits 15:0) of `mip`.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIE	0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 23. Standard portion (bits 15:0) of `mie`.

The machine-level interrupt registers handle a few root interrupt sources which are assigned a fixed service priority for simplicity, while separate external interrupt controllers can implement a more complex prioritization scheme over a much larger set of interrupts that are then multiplexed into the machine-level interrupt sources.

The non-maskable interrupt is not made visible via the `mip` register as its presence is implicitly known when executing the NMI trap handler.

Bits `mip.MEIP` and `mie.MEIE` are the interrupt-pending and interrupt-enable bits for machine-level external interrupts. `MEIP` is read-only in `mip`, and is set and cleared by a platform-specific interrupt controller.

Bits `mip.MTIP` and `mie.MTIE` are the interrupt-pending and interrupt-enable bits for machine timer interrupts. `MTIP` is read-only in the `mip` register, and is cleared by writing to the memory-mapped machine-mode timer compare register.

Bits `mip.MSIP` and `mie.MSIE` are the interrupt-pending and interrupt-enable bits for machine-level software interrupts. `MSIP` is read-only in `mip`, and is written by accesses to memory-mapped control registers, which are used to provide machine-level interprocessor interrupts.

A hart's memory-mapped `msip` register is a 32-bit read/write register, where bits 31—1 read as zero and bit 0 contains the `MSIP` bit. When the memory-mapped `msip` register changes, it is guaranteed to be reflected in `mip.MSIP` eventually, but not necessarily immediately. If a system has only one hart, or if a platform standard supports the delivery of machine-level interprocessor interrupts through external interrupts (MEI) instead, then `mip.MSIP` and `mie.MSIE` may both be read-only zeros.

If supervisor mode is not implemented, bits `SEIP`, `STIP`, and `SSIP` of `mip` and `SEIE`, `STIE`, and `SSIE` of `mie` are read-only zeros.

If supervisor mode is implemented, bits **mip.SEIP** and **mie.SEIE** are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in **mip**, and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software-writable SEIP bit and the signal from the external interrupt controller. When **mip** is read with a CSR instruction, the value of the SEIP bit returned in the **rd** destination register is the logical-OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRWS or CSRWC instruction.



*For example, if we name the software-writable SEIP bit **B** and the signal from the external interrupt controller **E**, then if **csrrs t0, mip, t1** is executed, **t0[9]** is written with **B || E**, then **B** is written with **B || t1[9]**. If **csrrw t0, mip, t1** is executed, then **t0[9]** is written with **B || E**, and **B** is simply written with **t1[9]**. In neither case does **B** depend upon **E**.*

The SEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts cleanly, without losing any real external interrupts. The behavior of the CSR instructions is slightly modified from regular CSR accesses as a result.

If supervisor mode is implemented, its **mip.STIP** and **mie.STIE** are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If the **stimecmp** register is not implemented, STIP is writable in **mip**, and may be written by M-mode software to deliver timer interrupts to S-mode. If the **stimecmp** (supervisor-mode timer compare) register is implemented, STIP is read-only in **mip** and reflects the supervisor-level timer interrupt signal resulting from **stimecmp**. This timer interrupt signal is cleared by writing **stimecmp** with a value greater than the current time value.

If supervisor mode is implemented, bits **mip.SSIP** and **mie.SSIE** are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in **mip** and may also be set to 1 by a platform-specific interrupt controller.

If the **Sscofpmf** extension is implemented, bits **mip.LCOFIP** and **mie.LCOFIE** are the interrupt-pending and interrupt-enable bits for local-counter-overflow interrupts. LCOFIP is read-write in **mip** and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the **mhpmeventn.OF** bits being set. If the **Sscofpmf** extension is not implemented, **mip.LCOFIP** and **mie.LCOFIE** are read-only zeros.

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

The machine-level interrupt fixed-priority ordering rules were developed with the following rationale.

Interrupts for higher privilege modes must be serviced before interrupts for lower privilege modes to support preemption.



The platform-specific machine-level interrupt sources in bits 16 and above have platform-specific priority, but are typically chosen to have the highest service priority to support very fast local vectored interrupts.

External interrupts are handled before internal (timer/software) interrupts as external interrupts are usually generated by devices that might require low interrupt service times.

Software interrupts are handled before internal timer interrupts, because internal timer interrupts are usually intended for time slicing, where time precision is less important,

whereas software interrupts are used for inter-processor messaging. Software interrupts can be avoided when high-precision timing is required, or high-precision timer interrupts can be routed via a different interrupt path. Software interrupts are located in the lowest four bits of **mip** as these are often written by software, and this position allows the use of a single CSR instruction with a five-bit immediate.

Restricted views of the **mip** and **mie** registers appear as the **sip** and **sie** registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the **mideleg** register, it becomes visible in the **sip** register and is maskable using the **sie** register. Otherwise, the corresponding bits in **sip** and **sie** are read-only zero.

3.1.10. Hardware Performance Monitor

M-mode includes a basic hardware performance-monitoring facility. The **mcycle** CSR counts the number of clock cycles executed by the processor core on which the hart is running. The **minstret** CSR counts the number of instructions the hart has retired. The **mcycle** and **minstret** registers have 64-bit precision on all RV32 and RV64 harts.

The counter registers have an arbitrary value after the hart is reset, and can be written with a given value. Any CSR write takes effect after the writing instruction has otherwise completed. The **mcycle** CSR may be shared between harts on the same core, in which case writes to **mcycle** will be visible to those harts. The platform should provide a mechanism to indicate which harts share an **mcycle** CSR.

The hardware performance monitor includes 29 additional 64-bit event counters, **mhpmcounter3-mhpmcounter31**. The event selector CSRs, **mhpmevent3-mhpmevent31**, are 64-bit WARL registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is defined to mean "no event." All counters should be implemented, but a legal implementation is to make both the counter and its corresponding event selector be read-only 0.

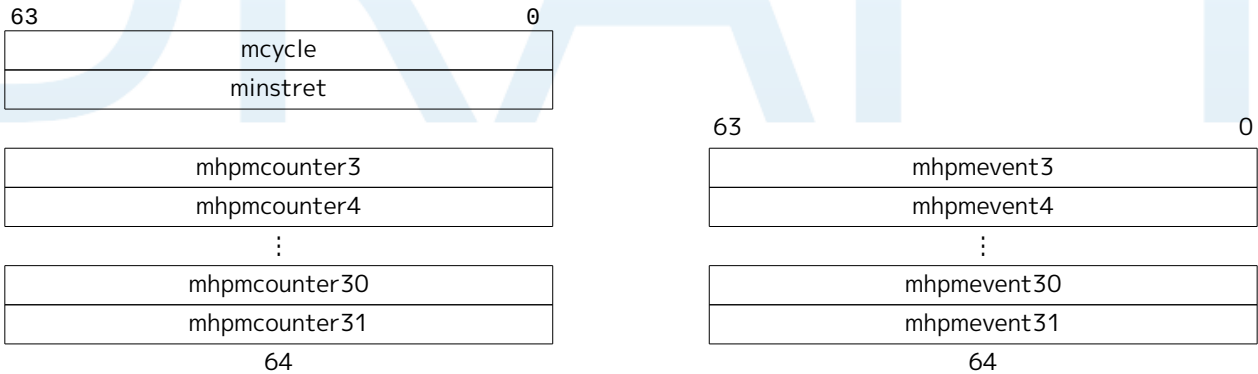


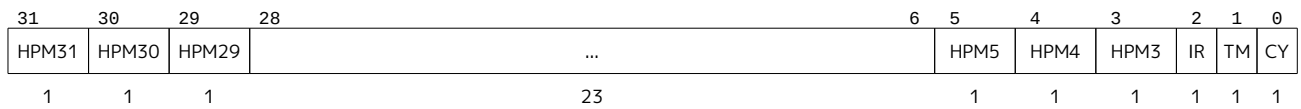
Figure 24. Hardware performance monitor counters.

The **mhpmcounters** are WARL registers that support up to 64 bits of precision on RV32 and RV64.

When **XLEN**=32, reads of the **mcycle**, **minstret**, **mhpmcountern**, and **mhpmeventn** CSRs return bitj 31-0 of the corresponding register, and writes change only bits 31-0; reads of the **mcycleh**, **minstreth**, **mhpmcounternh**, and **mhpmeventnh** CSRs return bits 63-32 of the corresponding register, and writes change only bits 63-32. The **mhpmeventnh** CSRs are provided only if the **Sscofpmf** extension is implemented.

3.1.11. Machine Counter-Enable (**mcounteren**) Register

The counter-enable **mcounteren** register is a 32-bit register that controls the availability of the hardware performance-monitoring counters to the next-lower privileged mode.

Figure 25. Counter-enable (**mcounteren**) register.

The settings in this register only control accessibility. The act of reading or writing this register does not affect the underlying counters, which continue to increment even when not accessible.

When the CY, TM, IR, or HPM_n bit in the **mcounteren** register is clear, attempts to read the **cycle**, **time**, **instret**, or **hpmcountern** register while executing in S-mode or U-mode will cause an illegal-instruction exception. When one of these bits is set, access to the corresponding register is permitted in the next implemented privilege mode (S-mode if implemented, otherwise U-mode).



The counter-enable bits support two common use cases with minimal hardware. For harts that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For harts that need high-performance timers and counters but are not concerned with obfuscating the underlying hardware counters, the counters can be directly exposed to lower privilege modes.

In addition, when the TM bit in the **mcounteren** register is clear, attempts to access the **stimecmp** or **vstimecmp** register while executing in a mode less privileged than M will cause an illegal-instruction exception. When this bit is set, access to the **stimecmp** or **vstimecmp** register is permitted in S-mode if implemented, and access to the **vstimecmp** register (via **stimecmp**) is permitted in VS-mode if implemented and not otherwise prevented by the TM bit in **hcounteren**.

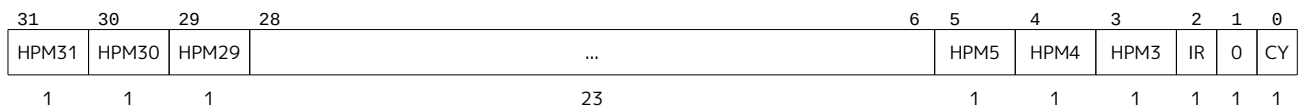
The **cycle**, **instret**, and **hpmcountern** CSRs are read-only shadows of **mcycle**, **minstret**, and **mhpmcountern**, respectively. The **time** CSR is a read-only shadow of the memory-mapped **mtime** register. Analogously, when XLEN=32, the **cycleh**, **instreth** and **hpmcounternh** CSRs are read-only shadows of **mcycleh**, **minstreth** and **mhpmcounternh**, respectively. When XLEN=32, the **timeh** CSR is a read-only shadow of the upper 32 bits of the memory-mapped **mtime** register, while **time** shadows only the lower 32 bits of **mtime**.



*Implementations can convert reads of the **time** and **timeh** CSRs into loads to the memory-mapped **mtime** register, or emulate this functionality on behalf of less-privileged modes in M-mode software.*

In harts with U-mode, the **mcounteren** must be implemented, but all fields are **WARL** and may be read-only zero, indicating reads to the corresponding counter will cause an illegal-instruction exception when executing in a less-privileged mode. In harts without U-mode, the **mcounteren** register should not exist.

3.1.12. Machine Counter-Inhibit (**mcountinhibit**) Register

Figure 26. Counter-inhibit **mcountinhibit** register

The counter-inhibit register **mcountinhibit** is a 32-bit **WARL** register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register.

When the CY, IR, or HPM_n bit in the **mcountinhibit** register is clear, the **mcycle**, **minstret**, or **mhpmcountern** register increments as usual. When the CY, IR, or HPM_n bit is set, the corresponding counter does not increment.

The **mcycle** CSR may be shared between harts on the same core, in which case the **mcountinhibit.CY** field is also shared between those harts, and so writes to **mcountinhibit.CY** will be visible to those harts.

If the **mcountinhibit** register is not implemented, the implementation behaves as though the register were set to zero.



When the **mcycle** and **minstret** counters are not needed, it is desirable to conditionally inhibit them to reduce energy consumption. Providing a single CSR to inhibit all counters also allows the counters to be atomically sampled.

Because the **mtime** counter can be shared between multiple cores, it cannot be inhibited with the **mcountinhibit** mechanism.

3.1.13. Machine Scratch (**mscratch**) Register

The **mscratch** register is an MXLEN-bit read/write register dedicated for use by machine mode. Typically, it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler.



Figure 27. Machine-mode scratch register.



The MIPS ISA allocated two user registers (**k0/k1**) for use by the operating system. Although the MIPS scheme provides a fast and simple implementation, it also reduces available user registers, and does not scale to further privilege levels, or nested traps. It can also require both registers are cleared before returning to user level to avoid a potential security hole and to provide deterministic debugging behavior.

The RISC-V user ISA was designed to support many possible privileged system environments and so we did not want to infect the user-level ISA with any OS-dependent features. The RISC-V CSR swap instructions can quickly save/restore values to the **mscratch** register. Unlike the MIPS design, the OS can rely on holding a value in the **mscratch** register while the user context is running.

3.1.14. Machine Exception Program Counter (**mepc**) Register

mepc is an MXLEN-bit read/write register formatted as shown in Figure 28. The low bit of **mepc** (**mepc[0]**) is always zero. On implementations that support only IALIGN=32, the two low bits (**mepc[1:0]**) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR **misal**, for example), then, whenever IALIGN=32, bit **mepc[1]** is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MRET instruction. Though masked, **mepc[1]** remains writable when IALIGN=32.

mepc is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing **mepc**, implementations may convert an invalid address into some other invalid address that **mepc** is capable of holding.



When address translation is not in effect, virtual addresses and physical addresses are equal. Hence, the set of addresses **mepc** must be able to represent includes the set of physical addresses that can be used as a valid **pc** or effective address.

When a trap is taken into M-mode, **mepc** is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, **mepc** is never written by the implementation, though it may be explicitly written by software.

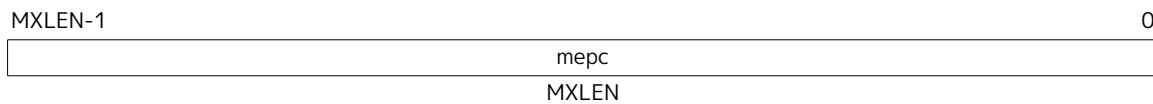


Figure 28. Machine exception program counter register.

3.1.15. Machine Cause (**mcause**) Register

The **mcause** register is an MXLEN-bit read-write register formatted as shown in Figure 29. When a trap is taken into M-mode, **mcause** is written with a code indicating the event that caused the trap. Otherwise, **mcause** is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the **mcause** register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Table 112 lists the possible machine-level exception codes. The Exception Code is a **WLRL** field, so is only guaranteed to hold supported exception codes.

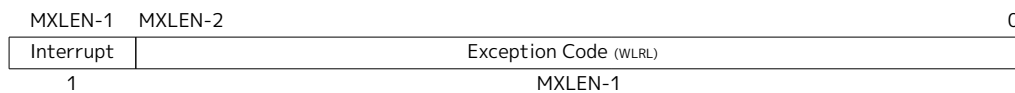


Figure 29. Machine Cause (**mcause**) register.

Note that load and load-reserved instructions generate load exceptions, whereas store, store-conditional, and AMO instructions generate store/AMO exceptions.

*Interrupts can be separated from other traps with a single branch on the sign of the **mcause** register value. A shift left can remove the interrupt bit and scale the exception codes to index into a trap vector table.*



We do not distinguish privileged instruction exceptions from illegal-instruction exceptions. This simplifies the architecture and also hides details of which higher-privilege instructions are supported by an implementation. The privilege level servicing the trap can implement a policy on whether these need to be distinguished, and if so, whether a given opcode should be treated as illegal or privileged.

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Table 113 indicates which exception is taken and reported in **mcause**. The priority of any custom synchronous exceptions is implementation-defined.

Table 112. Machine cause (**mcause**) register values after trap.

Interrupt	Exception Code	Description
1	0	<i>Reserved</i>
1	1	Supervisor software interrupt
1	2	<i>Reserved</i>
1	3	Machine software interrupt
1	4	<i>Reserved</i>
1	5	Supervisor timer interrupt
1	6	<i>Reserved</i>
1	7	Machine timer interrupt
1	8	<i>Reserved</i>
1	9	Supervisor external interrupt
1	10	<i>Reserved</i>
1	11	Machine external interrupt
1	12	<i>Reserved</i>
1	13	Counter-overflow interrupt
1	14-15	<i>Reserved</i>
1	≥16	<i>Designated for platform use</i>
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	<i>Reserved</i>
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16	Double trap
0	17	<i>Reserved</i>
0	18	Software check
0	19	Hardware error
0	20-23	<i>Reserved</i>
0	24-31	<i>Designated for custom use</i>
0	32-47	<i>Reserved</i>
0	48-63	<i>Designated for custom use</i>
0	≥64	<i>Reserved</i>

Table 113. Synchronous exception priority in decreasing priority order.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
	12, 1	During instruction address translation: First encountered page fault or access fault
	1	With physical address for instruction: Instruction access fault
	2 0 8,9,11 3 3	Illegal instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	4,6	Optionally: Load/store/AMO address misaligned
	13, 15, 5, 7	During address translation for an explicit memory access: First encountered page fault or access fault
	5,7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4,6	If not higher priority: Load/store/AMO address misaligned

When a virtual address is translated into a physical address, the address translation algorithm determines what specific exception may be raised.

Load/store/AMO address-misaligned exceptions may have either higher or lower priority than load/store/AMO page-fault and access-fault exceptions.

The relative priority of load/store/AMO address-misaligned and page-fault exceptions is implementation-defined to flexibly cater to two design points. Implementations that never support misaligned accesses can unconditionally raise the misaligned-address exception without performing address translation or protection checks. Implementations that support misaligned accesses only to some physical addresses must translate and check the address before determining whether the misaligned access may proceed, in which case raising the page-fault exception or access is more appropriate.



Instruction address breakpoints have the same cause value as, but different priority than, data address breakpoints (a.k.a. watchpoints) and environment break exceptions (which are raised by the EBREAK instruction).

Instruction address-misaligned exceptions are raised by control-flow instructions with misaligned targets, rather than by the act of fetching an instruction. Therefore, these exceptions have lower priority than other instruction address exceptions.



A software-check exception is a synchronous exception that is triggered when there are violations of checks and assertions defined by ISA extensions that aim to safeguard the integrity of software assets, including e.g. control-flow and memory-access constraints. When this exception is raised, the `xtval` register is set either to 0 or to an informative value defined

by the extension that stipulated the exception be raised. The priority of this exception, relative to other synchronous exceptions, depends on the cause of this exception and is defined by the extension that stipulated the exception be raised.

A hardware-error exception is a synchronous exception triggered when corrupted or uncorrectable data is accessed explicitly or implicitly by an instruction. In this context, "data" encompasses all types of information used within a RISC-V hart. Upon a hardware-error exception, the **xepc** register is set to the address of the instruction that attempted to access corrupted data, while the **xtval** register is set either to 0 or to the virtual address of an instruction fetch, load, or store that attempted to access corrupted data. The priority of hardware-error exception is implementation-defined, but any given occurrence is generally expected to be recognized at the point in the overall priority order at which the hardware error is discovered.

3.1.16. Machine Trap Value (**mtval**) Register

The **mtval** register is an MXLEN-bit read-write register formatted as shown in Figure 30. When a trap is taken into M-mode, **mtval** is either set to zero or written with exception-specific information to assist software in handling the trap. Otherwise, **mtval** is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set **mtval** informatively, which may unconditionally set it to zero, and which may exhibit either behavior, depending on the underlying event that caused the exception. If the hardware platform specifies that no exceptions set **mtval** to a nonzero value, then **mtval** is read-only zero.

If **mtval** is written with a nonzero value when a breakpoint, address-misaligned, access-fault, page-fault, or hardware-error exception occurs on an instruction fetch, load, or store, then **mtval** will contain the faulting virtual address.

On a breakpoint exception raised by an EBREAK or C.EBREAK instruction, **mtval** is written with either zero or the virtual address of the instruction.



For breakpoint exceptions raised by [C.]EBREAK, the virtual address of the instruction is already recorded in **mepc**. Recording the same address in **mtval** is redundant; the option is provided for backwards compatibility.

When page-based virtual memory is enabled, **mtval** is written with the faulting virtual address, even for physical-memory access-fault exceptions. This design reduces datapath cost for most implementations, particularly those with hardware page-table walkers.

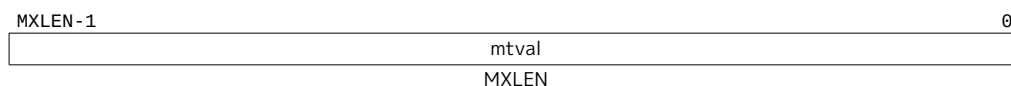


Figure 30. Machine Trap Value (**mtval**) register.

If **mtval** is written with a nonzero value when a misaligned load or store causes an access-fault, page-fault, or hardware-error exception, then **mtval** will contain the virtual address of the portion of the access that caused the fault.

If **mtval** is written with a nonzero value when an instruction access-fault, page-fault, or hardware-error exception occurs on a hart with variable-length instructions, then **mtval** will contain the virtual address of the portion of the instruction that caused the fault, while **mepc** will point to the beginning of the instruction.

The **mtval** register can optionally also be used to return the faulting instruction bits on an illegal-instruction exception (**mepc** points to the faulting instruction in memory). If **mtval** is written with a

nonzero value when an illegal-instruction exception occurs, then **mtval** will contain the shortest of: #

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first MXLEN bits of the faulting instruction

The value loaded into **mtval** on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

*Capturing the faulting instruction in **mtval** reduces the overhead of instruction emulation, potentially avoiding several partial instruction loads if the instruction is misaligned, and likely data cache misses or slow uncached accesses when loads are used to fetch the instruction into a data register. There is also a problem of atomicity if another agent is manipulating the instruction memory, as might occur in a dynamic translation system.*



*A requirement is that the entire instruction (or at least the first MXLEN bits) are fetched into **mtval** before taking the trap. This should not constrain implementations, which would typically fetch the entire instruction before attempting to decode the instruction, and avoids complicating software handlers.*

*A value of zero in **mtval** signifies either that the feature is not supported, or an illegal zero instruction was fetched. A load from the instruction memory pointed to by **mepc** can be used to distinguish these two cases (or alternatively, the system configuration information can be interrogated to install the appropriate trap handling before runtime).*

On a trap caused by a software-check exception, the **mtval** register holds the cause for the exception. The following encodings are defined:

- 0 - No information provided.
- 2 - Landing Pad Fault. Defined by the Zicfilp extension ([Section 6.9.1](#)).
- 3 - Shadow Stack Fault. Defined by the Zicfiss extension ([Section 6.9.2](#)).

For other traps, **mtval** is set to zero, but a future standard may redefine **mtval**'s setting for other traps.

If **mtval** is not read-only zero, it is a **WARL** register that must be able to hold all valid virtual addresses and the value zero. It need not be capable of holding all possible invalid addresses. Prior to writing **mtval**, implementations may convert an invalid address into some other invalid address that **mtval** is capable of holding. If the feature to return the faulting instruction bits is implemented, **mtval** must also be able to hold all values less than 2^N , where N is the smaller of MXLEN and ILEN.

3.1.17. Machine Configuration Pointer (**mconfigptr**) Register

The **mconfigptr** register is an MXLEN-bit read-only CSR formatted as shown in [Figure 31](#), that holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration.

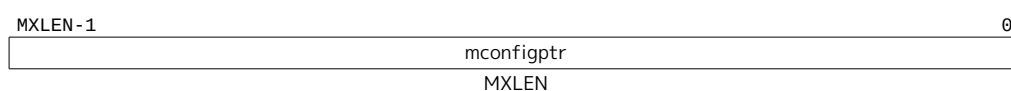


Figure 31. Machine Configuration Pointer (**mconfigptr**) register.

The pointer alignment in bits must be no smaller than MXLEN: i.e., if MXLEN is $8 \times n$, then **mconfigptr** $[\log_{2n}-1:0]$ must be zero.

mode and G-stage address translation. When `PBMTE=0`, the implementation behaves as though `Svpbmt` were not implemented. If `Svpbmt` is not implemented, `PBMTE` is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg.PBMTE` is read-only zero if `menvcfg.PBMTE` is zero.

After changing `menvcfg.PBMTE`, executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0` suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' `PBMT` fields. See [Section 5.5.3](#) for additional synchronization requirements when the hypervisor extension is implemented.

If the `Svadu` extension is implemented, the `ADUE` bit controls whether hardware updating of PTE A/D bits is enabled for S-mode and G-stage address translations. When `ADUE=1`, hardware updating of PTE A/D bits is enabled during S-mode address translation, and the implementation behaves as though the `Svade` extension were not implemented for S-mode address translation. When the hypervisor extension is implemented, if `ADUE=1`, hardware updating of PTE A/D bits is enabled during G-stage address translation, and the implementation behaves as though the `Svade` extension were not implemented for G-stage address translation. When `ADUE=0`, the implementation behaves as though `Svade` were implemented for S-mode and G-stage address translation. If `Svadu` is not implemented, `ADUE` is read-only zero. Furthermore, for implementations with the hypervisor extension, `henvcfg.ADUE` is read-only zero if `menvcfg.ADUE` is zero.

After changing `menvcfg.ADUE`, executing an `SFENCE.VMA` instruction with `rs1=x0` and `rs2=x0` suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' A/D bits. See [Section 5.5.3](#) for additional synchronization requirements when the hypervisor extension is implemented.



The Svade extension requires page-fault exceptions be raised when PTE A/D bits need be set, hence Svade is implemented when ADUE=0.

If the `Smcdeleg` extension is implemented, the `CDE` (Counter Delegation Enable) bit controls whether `Zicntr` and `Zihpm` counters can be delegated to S-mode. When `CDE=1`, the `Smcdeleg` extension is enabled, see [Section 6.6](#). When `CDE=0`, the `Smcdeleg` and `Scccfcfg` extensions appear to be not implemented. If `Smcdeleg` is not implemented, `CDE` is read-only zero.

The `Sstc` extension adds the `STCE` (STimecmp Enable) bit to `menvcfg` CSR. When the `Sstc` extension is not implemented, `STCE` is read-only zero. The `STCE` bit enables `stimecmp` for S-mode when set to one. When this extension is implemented and `STCE` in `menvcfg` is zero, an attempt to access `stimecmp` in a mode other than M-mode raises an illegal-instruction exception, `STCE` in `henvcfg` is read-only zero, and `STIP` in `mip` and `sip` reverts to its defined behavior as if this extension is not implemented. Further, if the H extension is implemented, then `hip.VSTIP` also reverts its defined behavior as if this extension is not implemented.

The `Zicboz` extension adds the `CBZE` (Cache Block Zero instruction enable) field to `menvcfg`. When the `CBZE` field is set to 1, it enables execution of the cache block zero instruction, `CB0.ZERO`, in modes less privileged than M. Otherwise, the instruction raises an illegal-instruction exception in modes less privileged than M. When the `Zicboz` extension is not implemented, `CBZE` is read-only zero.

The `Zicbom` extension adds the `CBCFE` (Cache Block Clean and Flush instruction Enable) field to `menvcfg`. When the `CBCFE` field is set to 1, it enables execution of the cache block clean instruction (`CB0.CLEAN`) and the cache block flush instruction (`CB0.FLUSH`) in modes less privileged than M. Otherwise, these instructions raise an illegal-instruction exception in modes less privileged than M. When the `Zicbom` extension is not implemented, `CBCFE` is read-only zero.

The `Zicbom` extension adds the `CBIE` (Cache Block Invalidate instruction Enable) WARL field to `menvcfg` to control execution of the cache block invalidate instruction (`CB0.INVALID`) in modes less privileged than M. When `CBIE` is set to `00b`, the instruction raises an illegal-instruction exception in modes less privileged than

M. When the Zicbom extension is not implemented, **CBIE** is read-only zero. The encoding **10b** is reserved. When **CBIE** is set to **01b** or **11b**, and when enabled for execution in modes less privileged than M, it behaves as follows:

- **01b** — The instruction is executed and performs a flush operation, even if configured by a mode less privileged than M to perform an invalidate operation.
- **11b** — The instruction is executed and performs an invalidate operation, unless configured by a mode less privileged than M to perform a flush operation.

If the Smnppm extension is implemented, the **PMM** field enables or disables pointer masking (see [Section 6.10](#)) for the next-lower privilege mode (S-/HS-mode if S-mode is implemented, or U-mode otherwise), according to the values in [Table 115](#). If Smnppm is not implemented, **PMM** is read-only zero. The **PMM** field is read-only zero for RV32.

Table 115. Legal values of **PMM** WARL field

Value	Description
00	Pointer masking is disabled (PMLen = 0)
01	Reserved
10	Pointer masking is enabled with PMLen = XLEN - 57 (PMLen = 7 on RV64)
11	Pointer masking is enabled with PMLen = XLEN - 48 (PMLen = 16 on RV64)

The Zicfilp extension adds the **LPE** field in **menvcfg**. When the **LPE** field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If **LPE** field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode. When the **LPE** field is 0, the Zicfilp extension is not enabled in S-mode, and the following rules apply to S-mode. If the **LPE** field is 0 and S-mode is not implemented, then the same rules apply to U-mode.

- The hart does not update the **ELP** state; it remains as **NO_LP_EXPECTED**.
- The **LPAD** instruction operates as a no-op.

The Zicfiss extension adds the **SSE** field to **menvcfg**. When the **SSE** field is set to 1 the Zicfiss extension is activated in S-mode. When **SSE** field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The **pte.xwr=010b** encoding in VS/S-stage page tables becomes reserved.
- **SSAMOSWAP.W/D** raises an illegal-instruction exception.

When **menvcfg.SSE** is 0, the **henvcfg.SSE** and **senvcfg.SSE** fields are read-only zero.

The Ssdbltrp extension adds the double-trap-enable (**DTE**) field in **menvcfg**. When **menvcfg.DTE** is zero, the implementation behaves as though Ssdbltrp is not implemented. When Ssdbltrp is not implemented **sstatus.SDT**, **vsstatus.SDT**, and **henvcfg.DTE** bits are read-only zero.

When XLEN=32, **menvcfgh** is a 32-bit read/write register that aliases bits 63:32 of **menvcfg**. The **menvcfgh** register does not exist when XLEN=64.

If U-mode is not supported, then registers **menvcfg** and **menvcfgh** do not exist.

3.1.19. Machine Security Configuration (mseccfg) Register

mseccfg is a 64-bit read/write register, formatted as shown in Figure 33, that controls security features. It exists if any extension that adds a field to **mseccfg** is implemented. Otherwise, it is reserved.

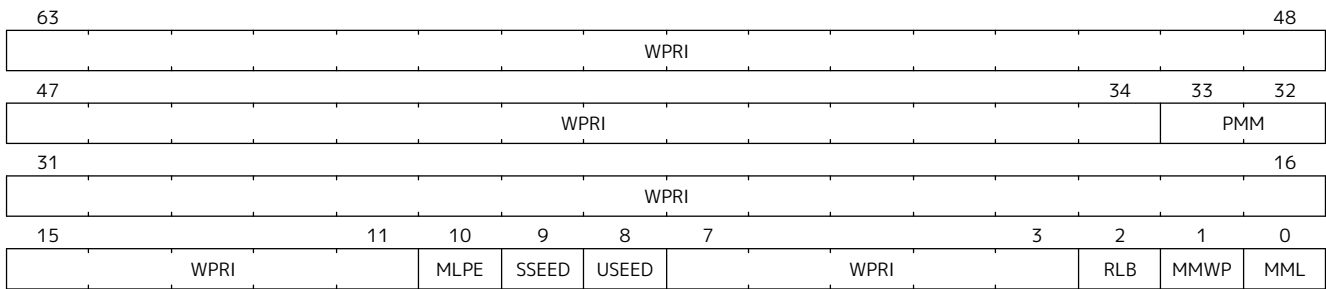


Figure 33. Machine security configuration (**mseccfg**) register.

The Zkr extension adds the **SSEED** and **USEED** fields to the **mseccfg** CSR to control access to the **seed** CSR from modes less privileged than M.

When **USEED** is 0, access to the **seed** CSR in U-mode raises an illegal-instruction exception. When **USEED** is 1, read-write access to the **seed** CSR from U-mode is allowed; all other types of accesses raise an illegal-instruction exception. If Zkr or U-mode is not implemented, **USEED** is read-only zero.

When **SSEED** is 0, access to the **seed** CSR from S-/HS-mode raises an illegal-instruction exception. When **SSEED** is 1, read-write access to the **seed** CSR from S-/HS-mode is allowed; all other types of accesses raise an illegal-instruction exception. If Zkr or S-mode is not implemented, **SSEED** is read-only zero.

When the H extension is also implemented, access to the **seed** CSR from an HS-qualified instruction leads to a virtual-instruction exception in VS and VU modes; all other types of accesses raise an illegal-instruction exception.

Table 116. Entropy Source Access Control.

Mode	SSEED	USEED	Description
M	-	-	The seed CSR is always available in machine mode as normal (with a CSR read-write instruction.) Attempted read without a write raises an illegal-instruction exception regardless of mode and access control bits.
U	-	0	Any seed CSR access raises an illegal-instruction exception.
U	-	1	The seed CSR is accessible as normal. No exception is raised for read-write.
S/HS	0	-	Any seed CSR access raises an illegal-instruction exception.
S/HS	1	-	The seed CSR is accessible as normal. No exception is raised for read-write.
VS/VU	0	-	Any seed CSR access raises an illegal-instruction exception.
VS/VU	1	-	A read-write seed access raises a virtual-instruction exception, while other access conditions raise an illegal-instruction exception.

The Smepmp extension adds the **RLB**, **MMWP**, and the **MML** fields in **mseccfg**.

When **mseccfg.RLB** (Rule Locking Bypass) a WARL field that provides a mechanism to temporarily modify **Locked** PMP rules. When **mseccfg.RLB** is 1, locked PMP rules may be removed or modified and locked PMP rules may be edited. When **mseccfg.RLB** is 0 and **pmppcf.L** is 1 in any rule or entry (including disabled entries), then **mseccfg.RLB** remains 0 and any further modifications to **mseccfg.RLB** are ignored until a PMP reset.



This feature is intended to be used as a debug mechanism, or as a temporary workaround

during the boot process for simplifying software, and optimizing the allocation of memory and PMP rules. Using this functionality under normal operation, after the boot process is completed, should be avoided since it weakens the protection of M-mode-only rules. Vendors who don't need this functionality may hardwire this field to 0.

The terminology used to specify the fields introduced by the Smepmp extension is listed in [Section 6.3](#).

The `mseccfg.MMWP` (Machine-Mode Allowlist Policy) is a WARL field. This field changes the default PMP policy for Machine mode when accessing memory regions that don't have a matching PMP rule. This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When set it changes the default PMP policy for M-mode when accessing memory regions that don't have a matching **PMP rule**, to **denied** instead of **ignored**.

The `mseccfg.MML` (Machine Mode Lockdown) is a WARL field. The **MML** bit changes the interpretation of the `pmpcf.L` bit defined in [Section 3.7.1.2](#). This is a sticky bit, meaning that once set it cannot be unset until a **PMP reset**. When `mseccfg.MML` is set the system's behavior changes in the following way:

1. The meaning of `pmpcf.L` changes: Instead of marking a rule as **locked** and **enforced** in all modes, it now marks a rule as **M-mode-only** when set and **S/U-mode-only** when unset. The formerly reserved encoding of `pmpcf.RW=01`, and the encoding `pmpcf.LRWX=1111`, now encode a **Shared-Region**.

An *M-mode-only* rule is **enforced** on Machine mode and **denied** in Supervisor or User mode. It also remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.

An *S/U-mode-only* rule is **enforced** on Supervisor and User modes and **denied** on Machine mode.

A *Shared-Region* rule is **enforced** on all modes, with restrictions depending on the `pmpcf.L` and `pmpcf.X` bits:

- A *Shared-Region* rule where `pmpcf.L` is not set can be used for sharing data between M-mode and S/U-mode, so is not executable. M-mode has read/write access to that region, and S/U-mode has read access if `pmpcf.X` is not set, or read/write access if `pmpcf.X` is set.
 - A *Shared-Region* rule where `pmpcf.L` is set can be used for sharing code between M-mode and S/U-mode, so is not writable. Both M-mode and S/U-mode have execute access on the region, and M-mode also has read access if `pmpcf.X` is set. The rule remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.
 - The encoding `pmpcf.LRWX=1111` can be used for sharing data between M-mode and S/U mode, where both modes only have read-only access to the region. The rule remains **locked** so that any further modifications to its associated configuration or address registers are ignored until a **PMP reset**, unless `mseccfg.RLB` is set.
2. Adding a rule with executable privileges that either is **M-mode-only** or a **locked Shared-Region** is not possible and such `pmpcf` writes are ignored, leaving `pmpcf` unchanged. This restriction can be temporarily lifted by setting `mseccfg.RLB` e.g. during the boot process.
 3. Executing code with Machine mode privileges is only possible from memory regions with a matching **M-mode-only** rule or a **locked Shared-Region** rule with executable privileges. Executing code from a region without a matching rule or with a matching *S/U-mode-only* rule is **denied**.
 4. If `mseccfg.MML` is not set, the combination of `pmpcf.RW=01` remains reserved for future standard use.

If the **Smmppm** extension is implemented, the **PMM** field enables or disables pointer masking (see [Section 6.10](#)) for M-mode according to the values in [Table 117](#). If **Smmppm** is not implemented, **PMM** is read-only zero. The **PMM** field is read-only zero for RV32.

Table 117. Legal values of **PMM** WARL field

Value	Description
00	Pointer masking is disabled ($PMLen = 0$)
01	Reserved
10	Pointer masking is enabled with $PMLen = XLen - 57$ ($PMLen = 7$ on RV64)
11	Pointer masking is enabled with $PMLen = XLen - 48$ ($PMLen = 16$ on RV64)



Smmppm implementations need to satisfy $\max(\text{largest supported virtual address size, largest supported supervisor physical address size}) \leftarrow (XLen - PMLen)$ bits to avoid any masking logic on the TLB access path.

The **Zicfilp** extension adds the **MLPE** field in **mseccfg**. When **MLPE** field is 1, **Zicfilp** extension is enabled in M-mode. When the **MLPE** field is 0, the **Zicfilp** extension is not enabled in M-mode and the following rules apply to M-mode.

- The hart does not update the **ELP** state; it remains as **NO_LP_EXPECTED**.
- The **LPAD** instruction operates as a no-op.

When $XLen=32$ only, **mseccfgh** is a 32-bit read/write register that aliases bits 63:32 of **mseccfg**. Register **mseccfgh** exists when $XLen=32$ and **mseccfg** is implemented; it does not exist when $XLen=64$.

3.2. Machine-Level Memory-Mapped Registers

3.2.1. Machine Timer (**mtime** and **mtimecmp**) Registers

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode read-write register, **mtime**. **mtime** must increment at constant frequency, and the platform must provide a mechanism for determining the period of an **mtime** tick. The **mtime** register will wrap around if the count overflows.

The **mtime** register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (**mtimecmp**). A machine timer interrupt becomes pending whenever **mtime** contains a value greater than or equal to **mtimecmp**, treating the values as unsigned integers. The interrupt remains posted until **mtimecmp** becomes greater than **mtime** (typically as a result of writing **mtimecmp**). The interrupt will only be taken if interrupts are enabled and the **MTIE** bit is set in the **mie** register.

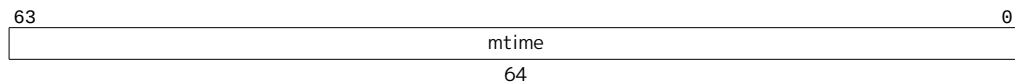


Figure 34. Machine time register (memory-mapped control register).

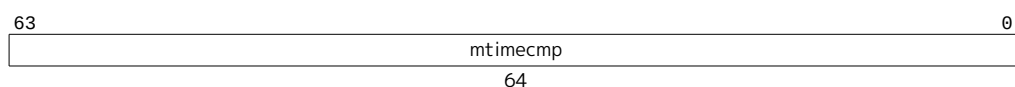


Figure 35. Machine time compare register (memory-mapped control register).



The timer facility is defined to use wall-clock time rather than a cycle counter to support modern processors that run with a highly variable clock frequency to save energy through

dynamic voltage and frequency scaling.

Accurate real-time clocks (RTCs) are relatively expensive to provide (requiring a crystal or MEMS oscillator) and have to run even when the rest of system is powered down, and so there is usually only one in a system located in a different frequency/voltage domain from the processors. Hence, the RTC must be shared by all the harts in a system and accesses to the RTC will potentially incur the penalty of a voltage-level-shifter and clock-domain crossing. It is thus more natural to expose `mtime` as a memory-mapped register than as a CSR.

Lower privilege levels do not have their own `timecmp` registers. Instead, machine-mode software can implement any number of virtual timers on a hart by multiplexing the next timer interrupt into the `mtimecmp` register.

Simple fixed-frequency systems can use a single clock for both cycle counting and wall-clock time.

If the result of the comparison between `mtime` and `mtimecmp` changes, it is guaranteed to be reflected in MTIP eventually, but not necessarily immediately.



A spurious timer interrupt might occur if an interrupt handler increments `mtimecmp` then immediately returns, because MTIP might not yet have fallen in the interim. All software should be written to assume this event is possible, but most software should assume this event is extremely unlikely. It is almost always more performant to incur an occasional spurious timer interrupt than to poll MTIP until it falls.

In RV32, memory-mapped writes to `mtimecmp` modify only one 32-bit part of the register. The following code sequence sets a 64-bit `mtimecmp` value without spuriously generating a timer interrupt due to the intermediate value of the comparand:

For RV64, naturally aligned 64-bit memory accesses to the `mtime` and `mtimecmp` registers are additionally supported and are atomic.

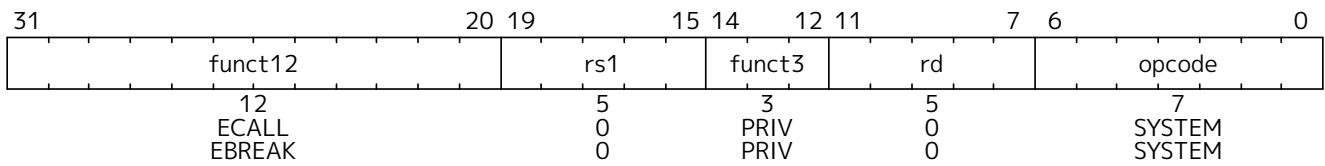
```
# New comparand is in a1:a0.
li t0, -1
la t1, mtimecmp
sw t0, 0(t1)      # No smaller than old value.
sw a1, 4(t1)      # No smaller than new value.
sw a0, 0(t1)      # New value.
```

Sample code for setting the 64-bit time comparand in RV32 assuming a little-endian memory system and that the registers live in a strongly ordered I/O region. Storing -1 to the low-order bits of `mtimecmp` prevents `mtimecmp` from temporarily becoming smaller than the lesser of the old and new values.

The `time` CSR is a read-only shadow of the memory-mapped `mtime` register. When XLEN=32, the `timeh` CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while `time` shadows only the lower 32 bits of `mtime`. When `mtime` changes, it is guaranteed to be reflected in `time` and `timeh` eventually, but not necessarily immediately.

3.3. Machine-Mode Privileged Instructions

3.3.1. Environment Call and Breakpoint



The ECALL instruction is used to make a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.



ECALL generates a different exception for each originating privilege mode so that environment call exceptions can be selectively delegated. A typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception but not the others.

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, EBREAK raises a breakpoint exception and performs no other operation.

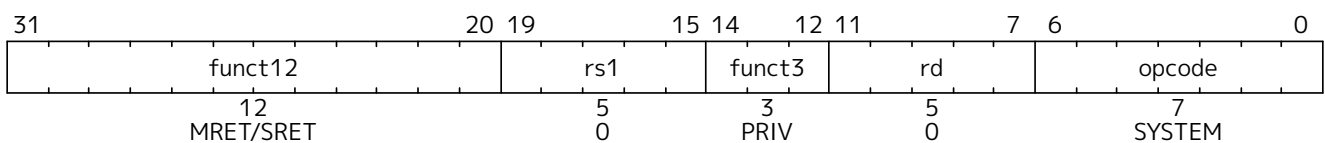


As described in [Volume I, Chapter 25](#), the C.EBREAK instruction performs the same operation as the EBREAK instruction.

ECALL and EBREAK cause the receiving privilege mode's **epc** register to be set to the address of the ECALL or EBREAK instruction itself, *not* the address of the following instruction. As ECALL and EBREAK cause synchronous exceptions, they are not considered to retire, and should not increment the **minstret** CSR.

3.3.2. Trap-Return Instructions

Instructions to return from trap are encoded under the PRIV minor opcode.



To return after handling a trap, there are separate trap return instructions per privilege level, MRET and SRET. MRET is always provided. SRET must be provided if supervisor mode is supported, and should raise an illegal-instruction exception otherwise. SRET should also raise an illegal-instruction exception when TSR=1 in **mstatus**, as described in [Section 3.1.6.6](#). An xRET instruction can be executed in privilege mode *x* or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. Attempting to execute an xRET instruction in a mode less privileged than *x* will raise an illegal-instruction exception.

In addition to manipulating the privilege stack as described in [Section 3.1.6.1](#), xRET sets the **pc** to the value stored in the **xepc** register.

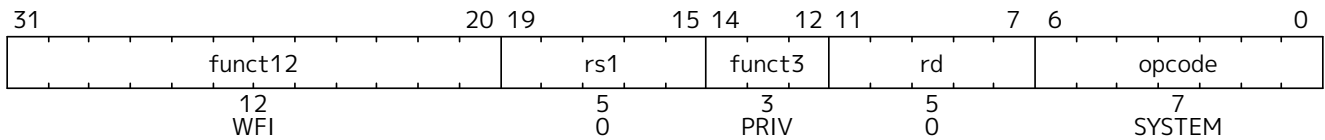
If the A extension is supported, the xRET instruction is allowed to clear any outstanding LR address reservation but is not required to. Trap handlers should explicitly clear the reservation if required (e.g., by using a dummy SC) before executing the xRET.



If xRET instructions always cleared LR reservations, it would be impossible to single-step through LR/SC sequences using a debugger.

3.3.3. Wait for Interrupt

The Wait for Interrupt instruction (WFI) informs the implementation that the current hart can be stalled until an interrupt might need servicing. Execution of the WFI instruction can also be used to inform the hardware platform that suitable interrupts should preferentially be routed to this hart. WFI is available in all privileged modes, and optionally available to U-mode. This instruction may raise an illegal-instruction exception when $TW=1$ in `mstatus`, as described in [Section 3.1.6.6](#).



If an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt trap will be taken on the following instruction, i.e., execution resumes in the trap handler and `mepc` = `pc` + 4.



The following instruction takes the interrupt trap so that a simple return from the trap handler will execute code after the WFI instruction.

Implementations are permitted to resume execution for any reason, even if an enabled interrupt has not become pending. Hence, a legal implementation is to simply implement the WFI instruction as a NOP.



If the implementation does not stall the hart on execution of the instruction, then the interrupt will be taken on some instruction in the idle loop containing the WFI, and on a simple return from the handler, the idle loop will resume execution.

The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in `mstatus` (MIE and SIE) and the delegation register `mideleg` (i.e., the hart must resume if a locally enabled interrupt becomes pending, even if it has been delegated to a less-privileged mode), but should honor the individual interrupt enables (e.g. MTIE) (i.e., implementations should avoid resuming the hart if the interrupt is pending but not individually enabled). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level.

If the event that causes the hart to resume execution does not cause an interrupt to be taken, execution will resume at `pc` + 4, and software must determine what action to take, including looping back to repeat the WFI if there was no actionable event.

By allowing wake-up when interrupts are disabled, an alternate entry point to an interrupt handler can be called that does not require saving the current context, as the current context can be saved or discarded before the WFI is executed.



As implementations are free to implement WFI as a NOP, software must explicitly check for any relevant pending but disabled interrupts in the code following an WFI, and should loop back to the WFI if no suitable interrupt was detected. The `mip` or `sip` registers can be interrogated to determine the presence of any interrupt in machine or supervisor mode respectively.

The operation of WFI is unaffected by the delegation register settings.

WFI is defined so that an implementation can trap into a higher privilege mode, either immediately on encountering the WFI or after some interval to initiate a machine-mode

transition to a lower power state, for example.

The same "wait-for-event" template might be used for possible future extensions that wait on memory locations changing, or message arrival.

3.3.4. Custom SYSTEM Instructions

The subspace of the SYSTEM major opcode shown in Figure 36 is designated for custom use. It is recommended that these instructions use bits 29:28 to designate the minimum required privilege mode, as do other SYSTEM instructions.

31	26	25	15	14	12	11	7	6	0	
funct6	custom			funct3	custom		opcode			Recommended Purpose
6	11			3	5		7			
100011	custom			0	custom		SYSTEM			Unprivileged or User-Level
110011	custom			0	custom		SYSTEM			Unprivileged or User-Level
100111	custom			0	custom		SYSTEM			Supervisor-Level
110111	custom			0	custom		SYSTEM			Supervisor-Level
101011	custom			0	custom		SYSTEM			HS-Level
111011	custom			0	custom		SYSTEM			HS-Level
101111	custom			0	custom		SYSTEM			Machine-Level
111111	custom			0	custom		SYSTEM			Machine-Level

Figure 36. SYSTEM instruction encodings designated for custom use.

3.4. Reset

Upon reset, a hart's privilege mode is set to M. The `mstatus` fields MIE and MPRV are reset to 0. If little-endian memory accesses are supported, the `mstatus/mstatush` field MBE is reset to 0. The `mis` register is reset to enable the maximal set of supported extensions, as described in Section 3.1.1. For implementations with the "A" standard extension, there is no valid load reservation. The `pc` is set to an implementation-defined reset vector. The `mcause` register is set to a value indicating the cause of the reset. Writable PMP registers' A and L fields are set to 0, unless the platform mandates a different reset value for some PMP registers' A and L fields. If the hypervisor extension is implemented, the `hgatp.MODE` and `vsatp.MODE` fields are reset to 0. If the Smrnmi extension is implemented, the `mnmstatus.NMIE` field is reset to 0. No WARL field contains an illegal value. If the Zicfilp extension is implemented, the `mseccfg.MLPE` field is reset to 0. All other hart state is UNSPECIFIED.

The `MML`, `MMWP`, and `RLB` fields of the `mseccfg` register are set to 0, unless the platform mandates a different reset value.

The `mcause` values after reset have implementation-specific interpretation, but the value 0 should be returned on implementations that do not distinguish different reset conditions. Implementations that distinguish different reset conditions should only use 0 to indicate the most complete reset.

The `USEED` and `SSEED` fields of the `mseccfg` CSR must have defined reset values. The system must not allow them to be in an undefined state after reset.



Some designs may have multiple causes of reset (e.g., power-on reset, external hard reset, brownout detected, watchdog timer elapse, sleep-mode wake-up), which machine-mode software and debuggers may wish to distinguish.

To avoid ambiguity, mcause reset values may alias mcause values following synchronous

*exceptions. There should be no ambiguity in this overlap, since on reset the **pc** is typically set to a different value than on other traps.*

3.5. Non-Maskable Interrupts

Non-maskable interrupts (NMIs) are only used for hardware error conditions, and cause an immediate jump to an implementation-defined NMI vector running in M-mode regardless of the state of a hart's interrupt enable bits. The **mepc** register is written with the virtual address of the instruction that was interrupted, and **mcause** is set to a value indicating the source of the NMI. The NMI can thus overwrite state in an active machine-mode interrupt handler.

The values written to **mcause** on an NMI are implementation-defined. The high Interrupt bit of **mcause** should be set to indicate that this was an interrupt. An Exception Code of 0 is reserved to mean "unknown cause" and implementations that do not distinguish sources of NMIs via the **mcause** register should return 0 in the Exception Code.

Unlike resets, NMIs do not reset processor state, enabling diagnosis, reporting, and possible containment of the hardware error.

3.6. Physical Memory Attributes

The physical memory map for a complete system includes various address ranges, some corresponding to memory regions and some to memory-mapped control registers, portions of which might not be accessible. Some memory regions might not support reads, writes, or execution; some might not support subword or subblock accesses; some might not support atomic operations; and some might not support cache coherence or might have different memory models. Similarly, memory-mapped control registers vary in their supported access widths, support for atomic operations, and whether read and write accesses have associated side effects. In RISC-V systems, these properties and capabilities of each region of the machine's physical address space are termed *physical memory attributes* (PMAs). This section describes RISC-V PMA terminology and how RISC-V systems implement and check PMAs.

PMAs are inherent properties of the underlying hardware and rarely change during system operation. Unlike physical memory protection values described in [Section 3.7](#), PMAs do not vary by execution context. The PMAs of some memory regions are fixed at chip design time—for example, for an on-chip ROM. Others are fixed at board design time, depending, for example, on which other chips are connected to off-chip buses. Off-chip buses might also support devices that could be changed on every power cycle (cold pluggable) or dynamically while the system is running (hot pluggable). Some devices might be configurable at run time to support different uses that imply different PMAs—for example, an on-chip scratchpad RAM might be cached privately by one core in one end-application, or accessed as a shared non-cached memory in another end-application.

Most systems will require that at least some PMAs are dynamically checked in hardware later in the execution pipeline after the physical address is known, as some operations will not be supported at all physical memory addresses, and some operations require knowing the current setting of a configurable PMA attribute. While many other architectures specify some PMAs in the virtual memory page tables and use the TLB to inform the pipeline of these properties, this approach injects platform-specific information into a virtualized layer and can cause system errors unless attributes are correctly initialized in each page-table entry for each physical memory region. In addition, the available page sizes might not be optimal for specifying attributes in the physical memory space, leading to address-space fragmentation and inefficient use of expensive TLB entries.

For RISC-V, we separate out specification and checking of PMAs into a separate hardware structure, the *PMA checker*. In many cases, the attributes are known at system design time for each physical address

region, and can be hardwired into the PMA checker. Where the attributes are run-time configurable, platform-specific memory-mapped control registers can be provided to specify these attributes at a granularity appropriate to each region on the platform (e.g., for an on-chip SRAM that can be flexibly divided between cacheable and uncacheable uses). PMAs are checked for any access to physical memory, including accesses that have undergone virtual to physical memory translation. To aid in system debugging, we strongly recommend that, where possible, RISC-V processors precisely trap physical memory accesses that fail PMA checks. Precisely trapped PMA violations manifest as instruction, load, or store access-fault exceptions, distinct from virtual-memory page-fault exceptions. Precise PMA traps might not always be possible, for example, when probing a legacy bus architecture that uses access failures as part of the discovery mechanism. In this case, error responses from peripheral devices will be reported as imprecise bus-error interrupts.

PMAs must also be readable by software to correctly access certain devices or to correctly configure other hardware components that access memory, such as DMA engines. As PMAs are tightly tied to a given physical platform's organization, many details are inherently platform-specific, as is the means by which software can learn the PMA values for a platform. Some devices, particularly legacy buses, do not support discovery of PMAs and so will give error responses or time out if an unsupported access is attempted. Typically, platform-specific machine-mode code will extract PMAs and ultimately present this information to higher-level less-privileged software using some standard representation.

Where platforms support dynamic reconfiguration of PMAs, an interface will be provided to set the attributes by passing requests to a machine-mode driver that can correctly reconfigure the platform. For example, switching cacheability attributes on some memory regions might involve platform-specific operations, such as cache flushes, that are available only to machine-mode.

3.6.1. Main Memory versus I/O Regions

The most important characterization of a given memory address range is whether it holds regular main memory or I/O devices. Regular main memory is required to have a number of properties, specified below, whereas I/O devices can have a much broader range of attributes. Memory regions that do not fit into regular main memory, for example, device scratchpad RAMs, are categorized as I/O regions.



What previous versions of this specification termed vacant regions are no longer a distinct category; they are now described as I/O regions that are not accessible (i.e. lacking read, write, and execute permissions). Main memory regions that are not accessible are also allowed.

3.6.2. Supported Access Type PMAs

Access types specify which access widths, from 8-bit byte to long multi-word burst, are supported, and also whether misaligned accesses are supported for each access width.



Although software running on a RISC-V hart cannot directly generate bursts to memory, software might have to program DMA engines to access I/O devices and might therefore need to know which access sizes are supported.

Main memory regions always support read and write of all access widths required by the attached devices, and can specify whether instruction fetch is supported.



Some platforms might mandate that all of main memory support instruction fetch. Other platforms might prohibit instruction fetch from some main memory regions.

In some cases, the design of a processor or device accessing main memory might support other

widths, but must be able to function with the types supported by the main memory.

I/O regions can specify which combinations of read, write, or execute accesses to which data widths are supported.

For systems with page-based virtual memory, I/O and memory regions can specify which combinations of hardware page-table reads and hardware page-table writes are supported.



Unix-like operating systems generally require that all of cacheable main memory supports page-table walks.

3.6.3. Atomicity PMAs

Atomicity PMAs describes which atomic instructions are supported in this address region. Support for atomic instructions is divided into two categories: LR/SC and AMOs.



Some platforms might mandate that all of cacheable main memory support all atomic operations required by the attached processors.

3.6.3.1. AMO PMA

Within AMOs, there are four levels of support: *AMONone*, *AMOSwap*, *AMOLogical*, and *AMOArithmetic*. *AMONone* indicates that no AMO operations are supported. *AMOSwap* indicates that only **amoswap** instructions are supported in this address range. *AMOLogical* indicates that swap instructions plus all the logical AMOs (**amoand**, **amoor**, **amoxor**) are supported. *AMOArithmetic* indicates that all RISC-V AMOs defined by the A extension are supported. For each level of support, naturally aligned AMOs of a given width are supported if the underlying memory region supports reads and writes of that width. Main memory and I/O regions may only support a subset or none of the processor-supported atomic operations.

Table 118. Classes of AMOs supported by I/O regions.

AMO Class	Supported Operations
AMONone	None
AMOSwap	amoswap
AMOLogical	above + amoand , amoor , amoxor
AMOArithmetic	above + amoadd , amomin , amomax , amominu , amomaxu



We recommend providing at least AMOLogical support for I/O regions where possible.

The Zacas extension defines three additional levels of support: *AMOCASW*, *AMOCASD*, and *AMOCASQ*.

AMOCASW indicates that in addition to instructions indicated by *AMOArithmetic* level support, the **AMOCAS.W** instruction is supported. *AMOCASD* indicates that in addition to instructions indicated by *AMOCASW* level support, the **AMOCAS.D** instruction is supported. *AMOCASQ* indicates that in addition to instructions indicated by *AMOCASD* level support, the **AMOCAS.Q** instruction is supported.



*AMOCASW/D/Q require AMOArithmetic level support as the **AMOCAS.W/D/Q** instructions require ability to perform an arithmetic comparison and a swap operation.*

The AMOs specified by the Zabha extension require the same level of support as the corresponding instructions in the A standard extension or the Zacas extension.

3.6.3.2. Reservability PMA

For LR/SC, there are three levels of support indicating combinations of the reservability and eventuality properties: *RsrvNone*, *RsrvNonEventual*, and *RsrvEventual*. *RsrvNone* indicates that no LR/SC operations are supported (the location is non-reservable). *RsrvNonEventual* indicates that the operations are supported (the location is reservable), but without the eventual success guarantee described in the unprivileged ISA specification. *RsrvEventual* indicates that the operations are supported and provide the eventual success guarantee.



We recommend providing RsrvEventual support for main memory regions where possible. Most I/O regions will not support LR/SC accesses, as these are most conveniently built on top of a cache-coherence scheme, but some may support RsrvNonEventual or RsrvEventual.

When LR/SC is used for memory locations marked RsrvNonEventual, software should provide alternative fall-back mechanisms used when lack of progress is detected.

3.6.4. Misaligned Atomicity Granule PMA

The misaligned atomicity granule PMA provides constrained support for misaligned AMOs. This PMA, if present, specifies the size of a *misaligned atomicity granule*, a naturally aligned power-of-two number of bytes. Specific supported values for this PMA are represented by MAGNN, e.g., MAG16 indicates the misaligned atomicity granule is at least 16 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.

If a misaligned AMO accesses a region that does not specify a misaligned atomicity granule PMA, or if not all accessed bytes lie within the same misaligned atomicity granule, then an exception is raised. For regular loads and stores that access such a region or for which not all accessed bytes lie within the same atomicity granule, then either an exception is raised, or the access proceeds but is not guaranteed to be atomic. Implementations may raise access-fault exceptions instead of address-misaligned exceptions for some misaligned accesses, indicating the instruction should not be emulated by a trap handler.



LR/SC instructions are unaffected by this PMA and so always raise an exception when misaligned. Vector memory accesses are also unaffected, so might execute non-atomically even when contained within a misaligned atomicity granule. Implicit accesses are similarly unaffected by this PMA.

3.6.5. Memory-Ordering PMAs

Regions of the address space are classified as either *main memory* or *I/O* for the purposes of ordering by the FENCE instruction and atomic-instruction ordering bits.

Accesses by one hart to main memory regions are observable not only by other harts but also by other devices with the capability to initiate requests in the main memory system (e.g., DMA engines). Coherent main memory regions always have either the RVWMO or RVTSO memory model. Incoherent main memory regions have an implementation-defined memory model.

Accesses by one hart to an I/O region are observable not only by other harts and bus mastering devices but also by the targeted I/O devices, and I/O regions may be accessed with either *relaxed* or *strong* ordering. Accesses to an I/O region with relaxed ordering are generally observed by other harts and bus mastering devices in a manner similar to the ordering of accesses to an RVWMO memory region, as discussed in the I/O Ordering section in [Volume I, Appendix B](#). By contrast, accesses to an I/O region with strong ordering are generally observed by other harts and bus mastering devices in program order.

Each strongly ordered I/O region specifies a numbered ordering channel, which is a mechanism by which ordering guarantees can be provided between different I/O regions. Channel 0 is used to indicate point-to-point strong ordering only, where only accesses by the hart to the single associated I/O region are strongly ordered.

Channel 1 is used to provide global strong ordering across all I/O regions. Any accesses by a hart to any I/O region associated with channel 1 can only be observed to have occurred in program order by all other harts and I/O devices, including relative to accesses made by that hart to relaxed I/O regions or strongly ordered I/O regions with different channel numbers. In other words, any access to a region in channel 1 is equivalent to executing a `fence io,io` instruction before and after the instruction.

Other larger channel numbers provide program ordering to accesses by that hart across any regions with the same channel number

Systems might support dynamic configuration of ordering properties on each memory region.

Strong ordering can be used to improve compatibility with legacy device driver code, or to enable increased performance compared to insertion of explicit ordering instructions when the implementation is known to not reorder accesses.

Local strong ordering (channel 0) is the default form of strong ordering as it is often straightforward to provide if there is only a single in-order communication path between the hart and the I/O device.

Generally, different strongly ordered I/O regions can share the same ordering channel without additional ordering hardware if they share the same interconnect path and the path does not reorder requests.

3.6.6. Coherence and Cacheability PMAs

Coherence is a property defined for a single physical address, and indicates that writes to that address by one agent will eventually be made visible to other coherent agents in the system. Coherence is not to be confused with the memory consistency model of a system, which defines what values a memory read can return given the previous history of reads and writes to the entire memory system. In RISC-V platforms, the use of hardware-incoherent regions is discouraged due to software complexity, performance, and energy impacts.

The cacheability of a memory region should not affect the software view of the region except for differences reflected in other PMAs, such as main memory versus I/O classification, memory ordering, supported accesses and atomic operations, and coherence. For this reason, we treat cacheability as a platform-level setting managed by machine-mode software only.

Where a platform supports configurable cacheability settings for a memory region, a platform-specific machine-mode routine will change the settings and flush caches if necessary, so the system is only incoherent during the transition between cacheability settings. This transitory state should not be visible to lower privilege levels.



Coherence is straightforward to provide for a shared memory region that is not cached by any

agent. The PMA for such a region would simply indicate it should not be cached in a private or shared cache.

Coherence is also straightforward for read-only regions, which can be safely cached by multiple agents without requiring a cache-coherence scheme. The PMA for this region would indicate that it can be cached, but that writes are not supported.

Some read-write regions might only be accessed by a single agent, in which case they can be cached privately by that agent without requiring a coherence scheme. The PMA for such regions would indicate they can be cached. The data can also be cached in a shared cache, as other agents should not access the region.

If an agent can cache a read-write region that is accessible by other agents, whether caching or non-caching, a cache-coherence scheme is required to avoid use of stale values. In regions lacking hardware cache coherence (hardware-incoherent regions), cache coherence can be implemented entirely in software, but software coherence schemes are notoriously difficult to implement correctly and often have severe performance impacts due to the need for conservative software-directed cache-flushing. Hardware cache-coherence schemes require more complex hardware and can impact performance due to the cache-coherence probes, but are otherwise invisible to software.

For each hardware cache-coherent region, the PMA would indicate that the region is coherent and which hardware coherence controller to use if the system has multiple coherence controllers. For some systems, the coherence controller might be an outer-level shared cache, which might itself access further outer-level cache-coherence controllers hierarchically.

Most memory regions within a platform will be coherent to software, because they will be fixed as either uncached, read-only, hardware cache-coherent, or only accessed by one agent.

If a PMA indicates non-cacheability, then accesses to that region must be satisfied by the memory itself, not by any caches.



For implementations with a cacheability-control mechanism, the situation may arise that a program uncacheably accesses a memory location that is currently cache-resident. In this situation, the cached copy must be ignored. This constraint is necessary to prevent more-privileged modes' speculative cache refills from affecting the behavior of less-privileged modes' uncacheable accesses.

3.6.7. Idempotency PMAs

Idempotency PMAs describe whether reads and writes to an address region are idempotent. Main memory regions are assumed to be idempotent. For I/O regions, idempotency on reads and writes can be specified separately (e.g., reads are idempotent but writes are not). If accesses are non-idempotent, i.e., there is potentially a side effect on any read or write access, then speculative or redundant accesses must be avoided.

For the purposes of defining the idempotency PMAs, changes in observed memory ordering created by redundant accesses are not considered a side effect.



While hardware should always be designed to avoid speculative or redundant accesses to memory regions marked as non-idempotent, it is also necessary to ensure software or compiler optimizations do not generate spurious accesses to non-idempotent memory regions.

Non-idempotent regions might not support misaligned accesses. Misaligned accesses to such regions should raise access-fault exceptions rather than address-misaligned exceptions,

indicating that software should not emulate the misaligned access using multiple smaller accesses, which could cause unexpected side effects.

For non-idempotent regions, implicit reads and writes must not be performed early or speculatively, with the following exceptions. When a non-speculative implicit read is performed, an implementation is permitted to additionally read any of the bytes within a naturally aligned power-of-2 region containing the address of the non-speculative implicit read. Furthermore, when a non-speculative instruction fetch is performed, an implementation is permitted to additionally read any of the bytes within the *next* naturally aligned power-of-2 region of the same size (with the address of the region taken modulo 2^{XLEN}). The results of these additional reads may be used to satisfy subsequent early or speculative implicit reads. The size of these naturally aligned power-of-2 regions is implementation-defined, but, for systems with page-based virtual memory, must not exceed the smallest supported page size.

3.7. Physical Memory Protection

To support secure processing and contain faults, it is desirable to limit the physical addresses accessible by software running on a hart. An optional physical memory protection (PMP) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The PMP values are checked in parallel with the PMA checks described in [Section 3.6](#).

The granularity of PMP access control settings are platform-specific, but the standard PMP encoding supports regions as small as four bytes. Certain regions' privileges can be hardwired—for example, some regions might only ever be visible in machine mode but in no lower-privilege layers.



Platforms vary widely in demands for physical memory protection, and some platforms may provide other PMP structures in addition to or instead of the scheme described in this section.

PMP checks are applied to all accesses whose effective privilege mode is S or U, including instruction fetches and data accesses in S and U mode, and data accesses in M-mode when the MPRV bit in `mstatus` is set and the MPP field in `mstatus` contains S or U. PMP checks are also applied to page-table accesses for virtual-address translation, for which the effective privilege mode is S. Optionally, PMP checks may additionally apply to M-mode accesses, in which case the PMP registers themselves are locked, so that even M-mode software cannot change them until the hart is reset. In effect, PMP can *grant* permissions to S and U modes, which by default have none, and can *revoke* permissions from M-mode, which by default has full permissions.

PMP violations are always trapped precisely at the processor.

3.7.1. Physical Memory Protection CSRs

PMP entries are described by an 8-bit configuration register and one MXLEN-bit address register. Some PMP settings additionally use the address register associated with the preceding PMP entry. Up to 64 PMP entries are supported. Implementations may implement zero, 16, or 64 PMP entries; the lowest-numbered PMP entries must be implemented first. All PMP CSR fields are **WARL** and may be read-only zero. PMP CSRs are only accessible to M-mode.

The PMP configuration registers are densely packed into CSRs to minimize context-switch time. For RV32, sixteen CSRs, `pmppc0cf0`–`pmppc0cf15`, hold the configurations `pmpp0cf0`–`pmpp63cf0` for the 64 PMP entries, as shown in [Figure 37](#). For RV64, eight even-numbered CSRs, `pmppc0cf0`, `pmppc2cf0`, ..., `pmppc14cf0`, hold the configurations for the 64 PMP entries, as shown in [Figure 38](#). For RV64, the odd-numbered configuration registers, `pmppc1cf0`, `pmppc3cf0`, ..., `pmppc15cf0`, are illegal.



RV64 harts use `pmppc2cf0`, rather than `pmppc1cf0`, to hold configurations for PMP entries 8-15.

This design reduces the cost of supporting multiple MXLEN values, since the configurations for PMP entries 8-11 appear in **pmpcfg2[31:0]** for both RV32 and RV64.

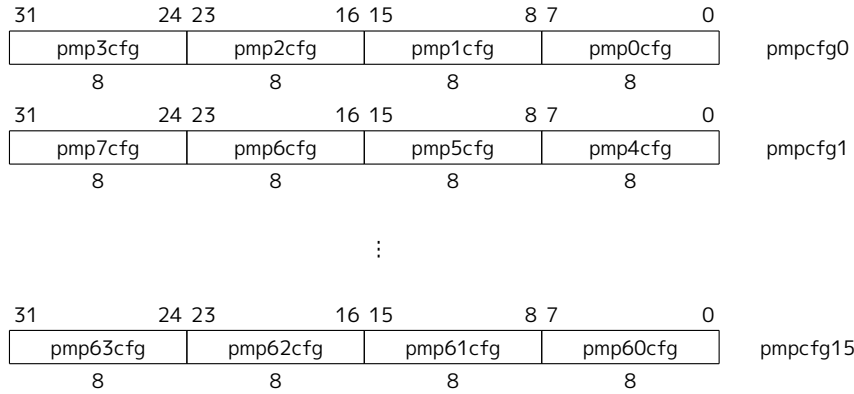


Figure 37. RV32 PMP configuration CSR layout.

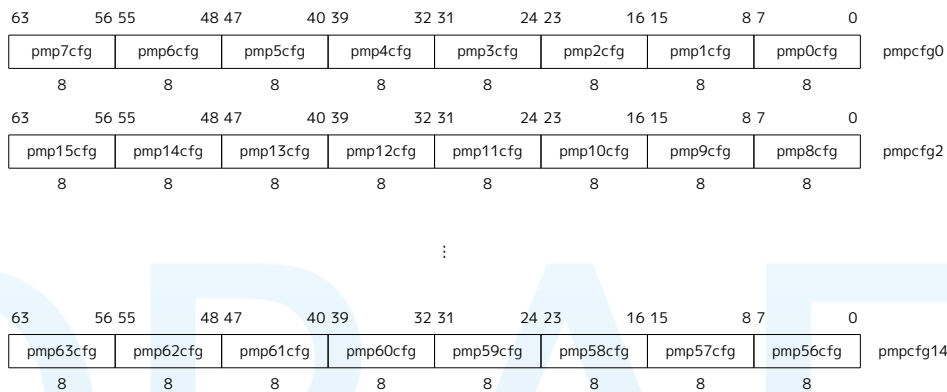


Figure 38. RV64 PMP configuration CSR layout.

The PMP address registers are CSRs named **pmpaddr0-pmpaddr63**. Each PMP address register encodes bits 33-2 of a 34-bit physical address for RV32, as shown in Figure 39. For RV64, each PMP address register encodes bits 55-2 of a 56-bit physical address, as shown in Figure 40. Not all physical address bits may be implemented, and so the **pmpaddr** registers are **WARL**.



The Sv32 page-based virtual-memory scheme described in Section 4.3 supports 34-bit physical addresses for RV32, so the PMP scheme must support addresses wider than XLEN for RV32. The Sv39 and Sv48 page-based virtual-memory schemes described in Section 4.4 and Section 4.5 support a 56-bit physical address space, so the RV64 PMP address registers impose the same limit.

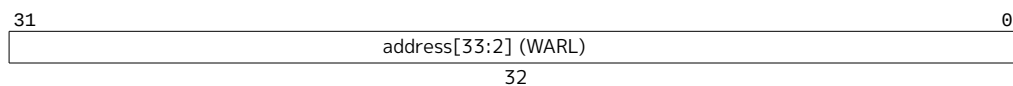


Figure 39. PMP address register format, RV32.



Figure 40. PMP address register format, RV64.

Figure 41 shows the layout of a PMP configuration register. The R, W, and X bits, when set, indicate that the PMP entry permits read, write, and instruction execution, respectively. When one of these bits is clear, the corresponding access type is denied. The R, W, and X fields form a collective **WARL** field for which the combinations with R=0 and W=1 are reserved. The remaining two fields, A and L, are described in the following sections.

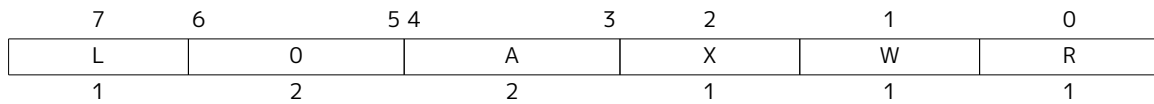


Figure 41. PMP configuration register format.

Attempting to fetch an instruction from a PMP region that does not have execute permissions raises an instruction access-fault exception. Attempting to execute a load, load-reserved, or cache-block management instruction which accesses a physical address within a PMP region without read permissions raises a load access-fault exception. Attempting to execute a store, store-conditional, AMO, or cache-block zero instruction which accesses a physical address within a PMP region without write permissions raises a store access-fault exception.

3.7.1.1. Address Matching

The A field in a PMP entry's configuration register encodes the address-matching mode of the associated PMP address register. The encoding of this field is shown in Table 119. When A=0, this PMP entry is disabled and matches no addresses. Two other address-matching modes are supported: naturally aligned power-of-2 regions (NAPOT), including the special case of naturally aligned four-byte regions (NA4); and the top boundary of an arbitrary range (TOR). These modes support four-byte granularity.

Table 119. Encoding of A field in PMP configuration registers.

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥ 8 bytes

NAPOT ranges make use of the low-order bits of the associated address register to encode the size of the range, as shown in Table 120.

Table 120. NAPOT range encoding in PMP address and configuration registers.

pmpaddr	pmpcfg.A	Match type and size
yyyy...yyyy	NA4	4-byte NAPOT range
yyyy...yyy0	NAPOT	8-byte NAPOT range
yyyy...yy01	NAPOT	16-byte NAPOT range
yyyy...y011	NAPOT	32-byte NAPOT range
...	...	...
yy01...1111	NAPOT	$2^{x_{LEN}}$ -byte NAPOT range
y011...1111	NAPOT	$2^{x_{LEN}+1}$ -byte NAPOT range
0111...1111	NAPOT	$2^{x_{LEN}+2}$ -byte NAPOT range
1111...1111	NAPOT	$2^{x_{LEN}+3}$ -byte NAPOT range

If TOR is selected, the associated address register forms the top of the address range, and the preceding PMP address register forms the bottom of the address range. If PMP entry i 's A field is set to TOR, the entry matches any address y such that $\text{pmpaddr}_{i-1} \leq y < \text{pmpaddr}_i$ (irrespective of the value of pmpcfg_{i-1}). If PMP entry 0's A field is set to TOR, zero is used for the lower bound, and so it matches any address $y < \text{pmpaddr}_0$.



If $\text{pmpaddr}_{i-1} \geq \text{pmpaddr}_i$ and $\text{pmpcfg}_i.A = \text{TOR}$, then PMP entry i matches no addresses.



It is not possible to represent the address $2^{x_{LEN}+2}$ as the top of a range, so, for example, in an RV32 system with 34-bit physical addresses, a TOR PMP cannot be used to give less-privileged modes access to the uppermost word of memory. Either a NAPOT PMP can be used, or that

memory can be left inaccessible to less-privileged modes. If the supervisor manages memory at base-page granularity, just 0.00002% of the 16 GiB address space is lost.

Although the PMP mechanism supports regions as small as four bytes, platforms may specify coarser PMP regions. In general, the PMP grain is 2^{G+2} bytes and must be the same across all PMP regions. When $G \geq 1$, the NA4 mode is not selectable. When $G \geq 2$ and `pmpcfgi.A[1]` is set, i.e. the mode is NAPOT, then bits `pmpaddri[G-2:0]` read as all ones. When $G \geq 1$ and `pmpcfgi.A[1]` is clear, i.e. the mode is OFF or TOR, then bits `pmpaddri[G-1:0]` read as all zeros. Bits `pmpaddri[G-1:0]` do not affect the TOR address-matching logic. Although changing `pmpcfgi.A[1]` affects the value read from `pmpaddri`, it does not affect the underlying value stored in that register—in particular, `pmpaddri[G-1]` retains its original value when `pmpcfgi.A` is changed from NAPOT to TOR/OFF then back to NAPOT.



Software may determine the PMP granularity by writing zero to `pmp0cfg`, then writing all ones to `pmpaddr0`, then reading back `pmpaddr0`. If G is the index of the least-significant bit set, the PMP granularity is 2^{G+2} bytes.

3.7.1.2. Locking and Privilege Mode

The L bit indicates that the PMP entry is locked, i.e., writes to the configuration register and associated address registers are ignored. Locked PMP entries remain locked until the hart is reset. If PMP entry i is locked, writes to `pmpicfg` and `pmpaddri` are ignored. Additionally, if PMP entry i is locked and `pmpicfg.A` is set to TOR, writes to `pmpaddri-1` are ignored.



Setting the L bit locks the PMP entry even when the A field is set to OFF.

In addition to locking the PMP entry, the L bit indicates whether the R/W/X permissions are additionally enforced on M-mode accesses. When the L bit is set, these permissions are enforced for all privilege modes. When the L bit is clear, any M-mode access matching the PMP entry will succeed; the R/W/X permissions apply only to S and U modes.

3.7.1.3. Priority and Matching Logic

On some implementations, misaligned loads, stores, and instruction fetches may be decomposed into multiple memory operations, some of which may succeed before an access-fault exception occurs, as described in the RVWMO specification. PMP checking is performed on each memory operation independently. In particular, a portion of a misaligned store that passes the PMP check may become visible, even if another portion fails the PMP check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

PMP entries are statically prioritized. The lowest-numbered PMP entry that matches any byte of a memory operation determines whether that operation succeeds or fails. The matching PMP entry must match all bytes of a memory operation, or the operation fails, irrespective of the L, R, W, and X bits. For example, if a PMP entry is configured to match the four-byte range `0xC–0xF`, then an 8-byte access to the range `0x8–0xF` will fail, assuming that PMP entry is the highest-priority entry that matches those addresses.

If a PMP entry matches all bytes of a memory operation, then the L, R, W, and X bits determine whether the operation succeeds or fails. If the L bit is clear and the privilege mode of the access is M, the operation succeeds. Otherwise, if the L bit is set or the privilege mode of the access is S or U, then the operation succeeds only if the R, W, or X bit corresponding to the access type is set.

If no PMP entry matches an M-mode memory operation, the operation succeeds. If no PMP entry matches an S-mode or U-mode memory operation, but at least one PMP entry is implemented, the operation fails.



If at least one PMP entry is implemented, but all PMP entries' A fields are set to OFF, then all

S-mode and U-mode memory accesses will fail.

Failed memory operations generate an instruction, load, or store access-fault exception. Note that a single instruction may generate multiple memory operations, which may not be mutually atomic. An access-fault exception is generated if at least one memory operation generated by an instruction fails, though other memory operations generated by that instruction may succeed with visible side effects. Notably, instructions that reference virtual memory are decomposed into multiple memory operations.

3.7.2. Physical Memory Protection and Paging

The Physical Memory Protection mechanism is designed to compose with the page-based virtual memory systems described in [Chapter 4](#). When paging is enabled, instructions that access virtual memory may result in multiple physical-memory accesses, including implicit references to the page tables. The PMP checks apply to all of these accesses. The effective privilege mode for implicit page-table accesses is S.

Implementations with virtual memory are permitted to perform address translations speculatively and earlier than required by an explicit memory access, and are permitted to cache them in address translation cache structures—including possibly caching the identity mappings from effective address to physical address used in Bare translation modes and M-mode. The PMP settings for the resulting physical address may be checked (and possibly cached) at any point between the address translation and the explicit memory access. Hence, when the PMP settings are modified, M-mode software must synchronize the PMP settings with the virtual memory system and any PMP or address-translation caches. This is accomplished by executing an SFENCE.VMA instruction with *rs1*=x0 and *rs2*=x0, after the PMP CSRs are written. See [Section 5.5.3](#) for additional synchronization requirements when the hypervisor extension is implemented.

If page-based virtual memory is not implemented, memory accesses check the PMP settings synchronously, so no SFENCE.VMA is needed.

Chapter 4. Supervisor-Level ISA, Version 1.13

This chapter describes the RISC-V supervisor-level architecture, which contains a common core that is used with various supervisor-level address translation and protection schemes.



Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization. In this spirit, certain supervisor-level facilities, including requests for timer and interprocessor interrupts, are provided by implementation-specific mechanisms. In some systems, a supervisor execution environment (SEE) provides these facilities in a manner specified by a supervisor binary interface (SBI). Other systems supply these facilities directly, through some other implementation-defined mechanism.

4.1. Supervisor CSRs

A number of CSRs are provided for the supervisor.



The supervisor should only view CSR state that should be visible to a supervisor-level operating system. In particular, there is no information about the existence (or non-existence) of higher privilege levels (machine level or other) visible in the CSRs accessible by the supervisor.

Many supervisor CSRs are a subset of the equivalent machine-mode CSR, and the machine-mode chapter should be read first to help understand the supervisor-level CSR descriptions.

4.1.1. Supervisor Status (sstatus) Register

The **sstatus** register is an SXLEN-bit read/write register formatted as shown in Figure 42 when SXLEN=32 and Figure 43 when SXLEN=64. The **sstatus** register keeps track of the processor's current operating state.



Figure 42. Supervisor-mode status (**sstatus**) register when SXLEN=32.

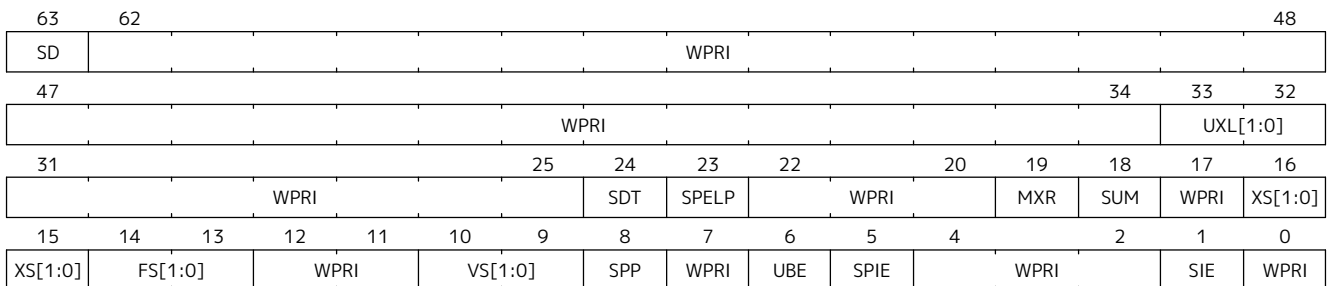


Figure 43. Supervisor-mode status (**sstatus**) register when SXLEN=64.

The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction (see Section 3.3.2) is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0.

The SIE bit enables or disables all interrupts in supervisor mode. When SIE is clear, interrupts are not taken while in supervisor mode. When the hart is running in user-mode, the value in SIE is ignored, and supervisor-level interrupts are enabled. The supervisor can disable individual interrupt sources using the

sie CSR.

The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1.

The **sstatus** register is a subset of the **mstatus** register.



*In a straightforward implementation, reading or writing any field in **sstatus** is equivalent to reading or writing the homonymous field in **mstatus**.*

4.1.1.1. Base ISA Control in **sstatus** Register

The UXL field controls the value of XLEN for U-mode, termed *UXLEN*, which may differ from the value of XLEN for S-mode, termed *SXLEN*. The encoding of UXL is the same as that of the MXL field of **misalr**, shown in [Table 107](#).

When *SXLEN*=32, the UXL field does not exist, and *UXLEN*=32. When *SXLEN*=64, it is a **WARL** field that encodes the current value of *UXLEN*. In particular, an implementation may make UXL be a read-only field whose value always ensures that *UXLEN*=*SXLEN*.

If *UXLEN*≠*SXLEN*, instructions executed in the narrower mode must ignore source register operand bits above the configured XLEN, and must sign-extend results to fill the widest supported XLEN in the destination register.

If *UXLEN* < *SXLEN*, user-mode instruction-fetch addresses and load and store effective addresses are taken modulo 2^{UXLEN} . For example, when *UXLEN*=32 and *SXLEN*=64, user-mode memory accesses reference the lowest 4 GiB of the address space.

Some HINT instructions are encoded as integer computational instructions that overwrite their destination register with its current value, e.g., **c.addi x8, 0**. When such a HINT is executed with *UXLEN* < *SXLEN* and bits *SXLEN*..*UXLEN* of the destination register not all equal to bit *UXLEN*-1, it is implementation-defined whether bits *SXLEN*..*UXLEN* of the destination register are unchanged or are overwritten with copies of bit *UXLEN*-1.



*This definition allows implementations to elide register write-back for some HINTs, while allowing them to execute other HINTs in the same manner as other integer computational instructions. The implementation choice is observable only by S-mode with *SXLEN* > *UXLEN*; it is invisible to U-mode.*

4.1.1.2. Memory Privilege in **sstatus** Register

The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1 in [Figure 60](#)) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed. MXR has no effect when page-based virtual memory is not in effect.

The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in [Figure 60](#)) will fault. When SUM=1, these accesses are permitted. SUM has no effect when page-based virtual memory is not in effect, nor when executing in U-mode. Note that S-mode can never execute instructions from user pages, regardless of the state of SUM.

SUM is read-only 0 if **satp.MODE** is read-only 0.



The SUM mechanism prevents supervisor software from inadvertently accessing user memory. Operating systems can execute the majority of code with SUM clear; the few code segments that should access user memory can temporarily set SUM.

The SUM mechanism does not avail S-mode software of permission to execute instructions in user code pages. Legitimate use cases for execution from user memory in supervisor context are rare in general and nonexistent in POSIX environments. However, bugs in supervisors that lead to arbitrary code execution are much easier to exploit if the supervisor exploit code can be stored in a user buffer at a virtual address chosen by an attacker.

Some non-POSIX single address space operating systems do allow certain privileged software to partially execute in supervisor mode, while most programs run in user mode, all in a shared address space. This use case can be realized by mapping the physical code pages at multiple virtual addresses with different permissions, possibly with the assistance of the instruction page-fault handler to direct supervisor software to use the alternate mapping.

4.1.1.3. Endianness Control in **sstatus** Register

The UBE bit is a WARL field that controls the endianness of explicit memory accesses made from U-mode, which may differ from the endianness of memory accesses in S-mode. An implementation may make UBE be a read-only field that always specifies the same endianness as for S-mode.

UBE controls whether explicit load and store memory accesses made from U-mode are little-endian (UBE=0) or big-endian (UBE=1).

UBE has no effect on instruction fetches, which are *implicit* memory accesses that are always little-endian.

For *implicit* accesses to supervisor-level memory management data structures, such as page tables, S-mode endianness always applies and UBE is ignored.



Standard RISC-V ABIs are expected to be purely little-endian-only or big-endian-only, with no accommodation for mixing endianness. Nevertheless, endianness control has been defined so as to permit an OS of one endianness to execute user-mode programs of the opposite endianness.

4.1.1.4. Previous Expected Landing Pad (ELP) State in **sstatus** Register

Access to the SPELP field, added by Zicfilp, accesses the homonymous fields of **mstatus** when **V=0**, and the homonymous fields of **vsstatus** when **V=1**.

4.1.1.5. Double Trap Control in **sstatus** Register

The S-mode-disable-trap (**SDT**) bit is a WARL field introduced by the Ssdbltrp extension to address double trap (See [Section 3.1.6.2](#)) at privilege modes lower than M.

When the **SDT** bit is set to 1 by an explicit CSR write, the **SIE** (Supervisor Interrupt Enable) bit is cleared to 0. This clearing occurs regardless of the value written, if any, to the **SIE** bit by the same write. The **SIE** bit can only be set to 1 by an explicit CSR write if the **SDT** bit is being set to 0 by the same write or is already 0.

When a trap is to be taken into S-mode, if the **SDT** bit is currently 0, it is then set to 1, and the trap is delivered as expected. However, if **SDT** is already set to 1, then this is an *unexpected trap*. In the event of an *unexpected trap*, a double-trap exception trap is delivered into M-mode. To deliver this trap, the hart writes registers, except **mcause** and **mtval2**, with the same information that the *unexpected trap* would have written if it was taken into M-mode. The **mtval2** register is then set to what would be otherwise written into the

mcause register by the *unexpected trap*. The **mcause** register is set to 16, the double-trap exception code.

An **SRET** instruction sets the **SDT** bit to 0.

*After a trap handler has saved the state, such as **scause**, **sepc**, and **stval**, needed for resuming from the trap and is reentrant, it should clear the **SDT** bit.*

*Resetting the **SDT** by an **SRET** enables the trap handler to detect a double trap that may occur during the tail phase, where it restores critical state to return from a trap.*

*The consequence of this specification is that if a critical error condition was caused by a guest-page fault, then the GPA will not be available in **mtval2** when the double trap is delivered to M-mode. This condition arises if the HS-mode invokes a hypervisor virtual-machine load or store instruction when **SDT** is 1 and the instruction raises a guest-page fault. The use of such an instruction in this phase of trap handling is not common. However, not recording the GPA is considered benign because, if required, it can still be obtained — albeit with added effort — through the process of walking the page tables.*

*For a double trap that originates in VS-mode, M-mode should redirect the exception to HS-mode by copying the values of M-mode CSRs updated by the trap to HS-mode CSRs and should use an **MRET** to resume execution at the address in **stvec**.*

Supervisor Software Events (SSE), an extension to the SBI, provide a mechanism for supervisor software to register and service system events emanating from an SBI implementation, such as firmware or a hypervisor. In the event of a double trap, HS-mode and M-mode can utilize the SSE mechanism to invoke a critical-error handler in VS-mode or S/HS-mode, respectively. Additionally, the implementation of an SSE protocol can be considered as an optional measure to aid in the recovery from such critical errors.

4.1.2. Supervisor Trap Vector Base Address (**stvec**) Register

The **stvec** register is an SXLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE).



Figure 44. Supervisor trap vector base address (**stvec**) register.

The BASE field in **stvec** is a field that can hold any valid virtual or physical address, subject to the following alignment constraints: the address must be 4-byte aligned, and MODE settings other than Direct might impose additional alignment constraints on the value in the BASE field.

Note that the CSR contains only bits XLEN-1 through 2 of the address BASE. When used as an address, the lower two bits are filled with zeroes to obtain an XLEN-bit address that is always aligned on a 4-byte boundary.

Table 121. Encoding of **stvec** MODE field.

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2		Reserved

The encoding of the MODE field is shown in Table 121. When MODE=Direct, all traps into supervisor mode cause the **pc** to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into supervisor mode cause the **pc** to be set to the address in the BASE field, whereas interrupts cause the **pc** to be set to the address in the BASE field plus four times the interrupt cause number. For example, a supervisor-mode timer interrupt (see Table 122) causes the **pc** to be set to $\text{BASE} + 0 \times 14$. Setting MODE=Vectored may impose a stricter alignment constraint on BASE.

4.1.3. Supervisor Interrupt (**sip** and **sie**) Registers

The **sip** register is an SXLEN-bit read/write register containing information on pending interrupts, while **sie** is the corresponding SXLEN-bit read/write register containing interrupt enable bits. Interrupt cause number i (as reported in CSR **scause**, Section 4.1.8) corresponds with bit i in both **sip** and **sie**. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.

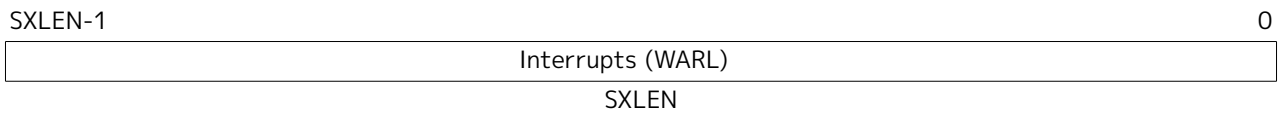


Figure 45. Supervisor interrupt-pending register (**sip**).

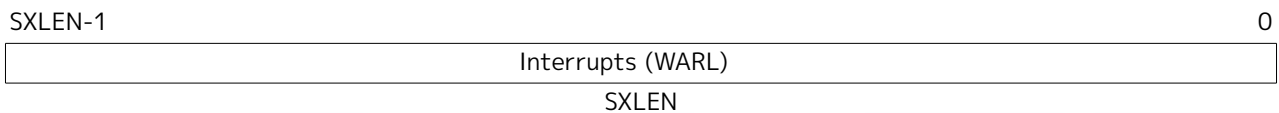


Figure 46. Supervisor interrupt-enable register (**sie**).

An interrupt i will trap to S-mode if both of the following are true: (a) either the current privilege mode is S and the SIE bit in the **sstatus** register is set, or the current privilege mode has less privilege than S-mode; and (b) bit i is set in both **sip** and **sie**.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in **sip**, and must also be evaluated immediately following the execution of an SRET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including **sip**, **sie** and **sstatus**).

Interrupts to S-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register **sip** may be writable or may be read-only. When bit i in **sip** is writable, a pending interrupt i can be cleared by writing 0 to this bit. If interrupt i can become pending but bit i in **sip** is read-only, the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment).

A bit in **sie** must be writable if the corresponding interrupt can ever become pending. Bits of **sie** that are not writable are read-only zero.

The standard portions (bits 15:0) of registers **sip** and **sie** are formatted as shown in Figures Figure 47 and Figure 48 respectively.

15	14	13	12	10	9	8	6	5	4	2	1	0
0	LCOFIP	0	SEIP	0	STIP	0	SSIP	0				
2	1	3	1	3	1	3	1	1				

Figure 47. Standard portion (bits 15:0) of **sip**.

15	14	13	12	10	9	8	6	5	4	2	1	0
0	LCOFIE	0	SEIE	0	STIE	0	SSIE	0				
2	1	3	1	3	1	3	1	1				

Figure 48. Standard portion (bits 15:0) of **sie**.

Bits **sip**.SEIP and **sie**.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. If implemented, SEIP is read-only in **sip**, and is set and cleared by the execution environment, typically through a platform-specific interrupt controller.

Bits **sip**.STIP and **sie**.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. If implemented, STIP is read-only in **sip**. When the Sstc extension is not implemented, STIP is set and cleared by the execution environment. When the Sstc extension is implemented, STIP reflects the timer interrupt signal resulting from **stimecmp**. The **sip**.STIP bit, in response to timer interrupts generated by **stimecmp**, is set by writing **stimecmp** with a value that is less than or equal to **time**, and is cleared by writing **stimecmp** with a value greater than **time**.

Bits **sip**.SSIP and **sie**.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. If implemented, SSIP is writable in **sip** and may also be set to 1 by a platform-specific interrupt controller.

If the Sscofpmf extension is implemented, bits **sip**.LCOFIP and **sie**.LCOFIE are the interrupt-pending and interrupt-enable bits for local-counter-overflow interrupts. LCOFIP is read-write in **sip** and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the **mhpmeventn**.OF bits being set. If the Sscofpmf extension is not implemented, **sip**.LCOFIP and **sie**.LCOFIE are read-only zeros.



*Interprocessor interrupts are sent to other harts by implementation-specific means, which will ultimately cause the SSIP bit to be set in the recipient hart's **sip** register.*

Each standard interrupt type (SEI, STI, SSI, or LCOFI) may not be implemented, in which case the corresponding interrupt-pending and interrupt-enable bits are read-only zeros. All bits in **sip** and **sie** are WARL fields. The implemented interrupts may be found by writing one to every bit location in **sie**, then reading back to see which bit positions hold a one.



*The **sip** and **sie** registers are subsets of the **mip** and **mie** registers. Reading any implemented field, or writing any writable field, of **sip**/**sie** effects a read or write of the homonymous field of **mip**/**mie**.*

*Bits 3, 7, and 11 of **sip** and **sie** correspond to the machine-mode software, timer, and external interrupts, respectively. Since most platforms will choose not to make these interrupts delegatable from M-mode to S-mode, they are shown as 0 in Figure 47 and Figure 48.*

Multiple simultaneous interrupts destined for supervisor mode are handled in the following decreasing priority order: SEI, SSI, STI, LCOFI.

4.1.4. Supervisor Timers and Performance Counters

Supervisor software uses the same hardware performance monitoring facility as user-mode software, including the **time**, **cycle**, and **instret** CSRs. The implementation should provide a mechanism to modify the counter values.

The implementation must provide a facility for scheduling timer interrupts in terms of the real-time counter, **time**.

4.1.5. Counter-Enable (**scounteren**) Register

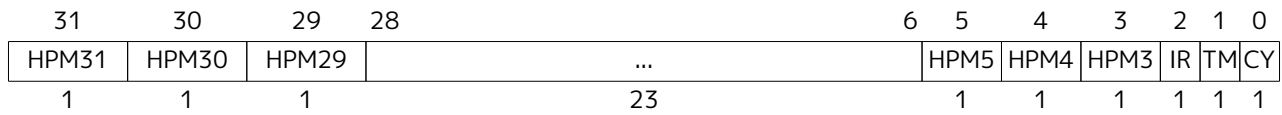


Figure 49. Counter-enable (**scounteren**) register

The counter-enable (**scounteren**) CSR is a 32-bit register that controls the availability of the hardware performance monitoring counters to U-mode.

When the CY, TM, IR, or HPM n bit in the **scounteren** register is clear, attempts to read the **cycle**, **time**, **instret**, or **hpmcountern** register while executing in U-mode will cause an illegal-instruction exception. When one of these bits is set, access to the corresponding register is permitted.

scounteren must be implemented. However, any of the bits may be read-only zero, indicating reads to the corresponding counter will cause an exception when executing in U-mode. Hence, they are effectively WARL fields.



*The setting of a bit in **mcouteren** does not affect whether the corresponding bit in **scounteren** is writable. However, U-mode may only access a counter if the corresponding bits in **scounteren** and **mcouteren** are both set.*

4.1.6. Supervisor Scratch (**sscratch**) Register

The **sscratch** CSR is an SXLEN-bit read/write register, dedicated for use by the supervisor. Typically, **sscratch** is used to hold a pointer to the hart-local supervisor context while the hart is executing user code. At the beginning of a trap handler, software normally uses a CSRRW instruction to swap **sscratch** with an integer register to obtain an initial working register.



Figure 50. Supervisor Scratch Register

4.1.7. Supervisor Exception Program Counter (**sepc**) Register

sepc is an SXLEN-bit read/write CSR formatted as shown in Figure 51. The low bit of **sepc** (**sepc**[0]) is always zero. On implementations that support only IALIGN=32, the two low bits (**sepc**[1:0]) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR **mis**a, for example), then, whenever IALIGN=32, bit **sepc**[1] is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the SRET instruction. Though masked, **sepc**[1] remains writable when IALIGN=32.

sepc is a WARL register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing **sepc**, implementations may convert an invalid address into some other invalid address that **sepc** is capable of holding.

When a trap is taken into S-mode, **sepc** is written with the virtual address of the instruction that was interrupted or that encountered the exception. Otherwise, **sepc** is never written by the implementation, though it may be explicitly written by software.

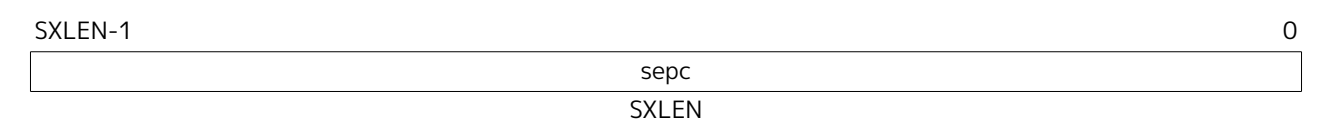


Figure 51. Supervisor exception program counter register.

4.1.8. Supervisor Cause (scause) Register

The **scause** CSR is an SXLEN-bit read-write register formatted as shown in [Figure 52](#). When a trap is taken into S-mode, **scause** is written with a code indicating the event that caused the trap. Otherwise, **scause** is never written by the implementation, though it may be explicitly written by software.

The Interrupt bit in the **scause** register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. [Table 122](#) lists the possible exception codes for the current supervisor ISAs. The Exception Code is a **WLRL** field. It is required to hold the values 0–31 (i.e., bits 4–0 must be implemented), but otherwise it is only guaranteed to hold supported exception codes.



Figure 52. Supervisor Cause (scause) register.

Table 122. Supervisor cause (scause) register values after trap. Synchronous exception priorities are given by [Table 113](#).

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2-4	Reserved
1	5	Supervisor timer interrupt
1	6-8	Reserved
1	9	Supervisor external interrupt
1	10-12	Reserved
1	13	Counter-overflow interrupt
1	14-15	Reserved
1	≥16	Designated for platform use

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10-11	<i>Reserved</i>
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16-17	<i>Reserved</i>
0	18	Software check
0	19	Hardware error
0	20-23	<i>Reserved</i>
0	24-31	<i>Designated for custom use</i>
0	32-47	<i>Reserved</i>
0	48-63	<i>Designated for custom use</i>
0	≥ 64	<i>Reserved</i>

4.1.9. Supervisor Trap Value (stval) Register

The **stval** CSR is an SXLEN-bit read-write register formatted as shown in Figure 53. When a trap is taken into S-mode, **stval** is written with exception-specific information to assist software in handling the trap. Otherwise, **stval** is never written by the implementation, though it may be explicitly written by software. The hardware platform will specify which exceptions must set **stval** informatively, which may unconditionally set it to zero, and which may exhibit either behavior, depending on the underlying event that caused the exception.

If **stval** is written with a nonzero value when a breakpoint, address-misaligned, access-fault, page-fault, or hardware-error exception occurs on an instruction fetch, load, or store, then **stval** will contain the faulting virtual address.

On a breakpoint exception raised by an EBREAK or C.EBREAK instruction, **stval** is written with either zero or the virtual address of the instruction.

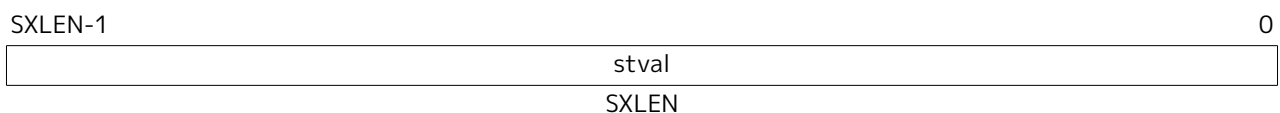


Figure 53. Supervisor Trap Value register.

If **stval** is written with a nonzero value when a misaligned load or store causes an access-fault, page-fault, or hardware-error exception, then **stval** will contain the virtual address of the portion of the access that caused the fault.

If **stval** is written with a nonzero value when an instruction access-fault, page-fault, or hardware-error

exception occurs on a hart with variable-length instructions, then **stval** will contain the virtual address of the portion of the instruction that caused the fault, while **sepc** will point to the beginning of the instruction.

The **stval** register can optionally also be used to return the faulting instruction bits on an illegal-instruction exception (**sepc** points to the faulting instruction in memory). If **stval** is written with a nonzero value when an illegal-instruction exception occurs, then **stval** will contain the shortest of:

- the actual faulting instruction
- the first ILEN bits of the faulting instruction
- the first SXLEN bits of the faulting instruction

The value loaded into **stval** on an illegal-instruction exception is right-justified and all unused upper bits are cleared to zero.

On a trap caused by a software-check exception, the **stval** register holds the cause for the exception. The following encodings are defined:

- 0 - No information provided.
- 2 - Landing Pad Fault. Defined by the Zicfilp extension ([Section 6.9.1](#)).
- 3 - Shadow Stack Fault. Defined by the Zicfiss extension ([Section 6.9.2](#)).

For other traps, **stval** is set to zero, but a future standard may redefine **stval**'s setting for other traps.

stval is a WARL register that must be able to hold all valid virtual addresses and the value 0. It need not be capable of holding all possible invalid addresses. Prior to writing **stval**, implementations may convert an invalid address into some other invalid address that **stval** is capable of holding. If the feature to return the faulting instruction bits is implemented, **stval** must also be able to hold all values less than 2^N , where N is the smaller of SXLEN and ILEN.

4.1.10. Supervisor Environment Configuration (**senvcfg**) Register

The **senvcfg** CSR is an SXLEN-bit read/write register, formatted as shown in [Figure 54](#), that controls certain characteristics of the U-mode execution environment.

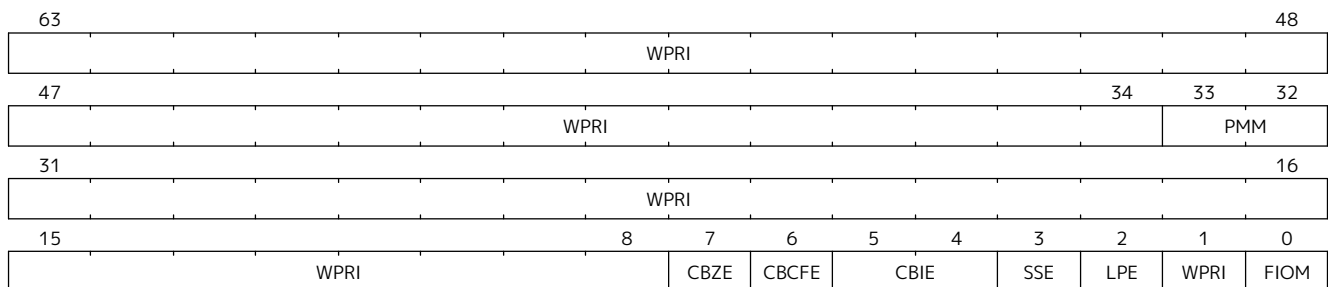


Figure 54. Supervisor environment configuration register (**senvcfg**) for RV64.

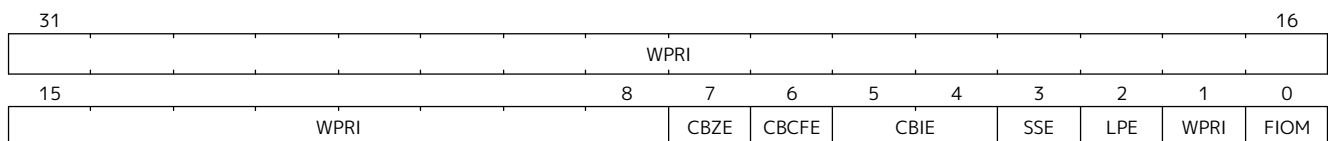


Figure 55. Supervisor environment configuration register (**senvcfg**) for RV32.

If bit FIOM (Fence of I/O implies Memory) is set to one in **senvcfg**, FENCE instructions executed in U-mode are modified so the requirement to order accesses to device I/O implies also the requirement to order

main memory accesses. [Table 123](#) details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO in U-mode when FIOM=1.

Similarly, for U-mode when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

If **satp.MODE** is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 123. Modified interpretation of FENCE predecessor and successor sets in U-mode when FIOM=1.

Instruction bit	Meaning when set
PI PO	Predecessor device input and memory reads (PR implied) Predecessor device output and memory writes (PW implied)
SI SO	Successor device input and memory reads (SR implied) Successor device output and memory writes (SW implied)

Bit FIOM exists for a specific circumstance when an I/O device is being emulated for U-mode and both of the following are true: (a) the emulated device has a memory buffer that should be I/O space but is actually mapped to main memory via address translation, and (b) multiple physical harts are involved in accessing this emulated device from U-mode.

A hypervisor running in S-mode without the benefit of the hypervisor extension of [Chapter 5](#) may need to emulate a device for U-mode if paravirtualization cannot be employed. If the same hypervisor provides a virtual machine (VM) with multiple virtual harts, mapped one-to-one to real harts, then multiple harts may concurrently access the emulated device, perhaps because: (a) the guest OS within the VM assigns device interrupt handling to one hart while the device is also accessed by a different hart outside of an interrupt handler, or (b) control of the device (or partial control) is being migrated from one hart to another, such as for interrupt load balancing within the VM. For such cases, guest software within the VM is expected to properly coordinate access to the (emulated) device across multiple harts using mutex locks and/or interprocessor interrupts as usual, which in part entails executing I/O fences. But those I/O fences may not be sufficient if some of the device “I/O” is actually main memory, unknown to the guest. Setting FIOM=1 modifies those fences (and all other I/O fences executed in U-mode) to include main memory, too.

Software can always avoid the need to set FIOM by never using main memory to emulate a device memory buffer that should be I/O space. However, this choice usually requires trapping all U-mode accesses to the emulated buffer, which might have a noticeable impact on performance. The alternative offered by FIOM is sufficiently inexpensive to implement that we consider it worth supporting even if only rarely enabled.

The Zicboz extension adds the **CBZE** (Cache Block Zero instruction enable) field to **senvcfg**. The **CBZE** field controls execution of the cache block zero instruction (**CB0.ZERO**) in U-mode. Execution of **CB0.ZERO** in U-mode is enabled only if execution of the instruction is enabled for use in S-mode and **CBZE** is set to 1; otherwise, an illegal-instruction exception is raised. When the Zicboz extension is not implemented, **CBZE** is read-only zero.

The Zicbom extension adds the **CBCFE** (Cache Block Clean and Flush instruction Enable) field to **senvcfg** to control execution of the **CB0.CLEAN** and **CB0.FLUSH** instructions in U-mode. Execution of these instructions in U-mode is enabled only if execution of these instructions is enabled for use in S-mode and **CBCFE** is set to 1; otherwise, an illegal-instruction exception is raised. When the Zicbom extension is not implemented, **CBCFE** is read-only zero.

The Zicbom extension adds the **CBIE** (Cache Block Invalidate instruction Enable) WARL field to **senvcfg** to

control execution of the **CB0.INVAL** instruction in U-mode. The encoding **10b** is reserved. When the Zicbom extension is not implemented, **CBIE** is read-only zero. Execution of **CB0.INVAL** in U-mode is enabled only if execution of the instruction is enabled for use in S-mode and **CBIE** is set to **01b** or **11b**; otherwise, an illegal-instruction exception is raised.

If **CB0.INVAL** is enabled in S-mode to perform a flush operation, then when the instruction is enabled in U-mode it performs a flush operation, even if **CBIE** is set to **11b**. Otherwise, the instruction behaves as follows, depending on the **CBIE** encoding:

- **01b** — The instruction is executed and performs a flush operation.
- **11b** — The instruction is executed and performs an invalidate operation.

If the Ssnpm extension is implemented, the **PMM** field enables or disables pointer masking (see [Section 6.10](#)) for the next-lower privilege mode (U/VU), according to the values in [Table 124](#). If Ssnpm is not implemented, **PMM** is read-only zero. The **PMM** field is read-only zero for RV32.

Table 124. Legal values of **PMM** WARL field

Value	Description
00	Pointer masking is disabled (PMLen = 0)
01	Reserved
10	Pointer masking is enabled with PMLen = XLEN - 57 (PMLen = 7 on RV64)
11	Pointer masking is enabled with PMLen = XLEN - 48 (PMLen = 16 on RV64)

The Zicfilp extension adds the **LPE** field in **senvcfg**. When the **LPE** field is set to 1, the Zicfilp extension is enabled in VU/U-mode. When the **LPE** field is 0, the Zicfilp extension is not enabled in VU/U-mode and the following rules apply to VU/U-mode:

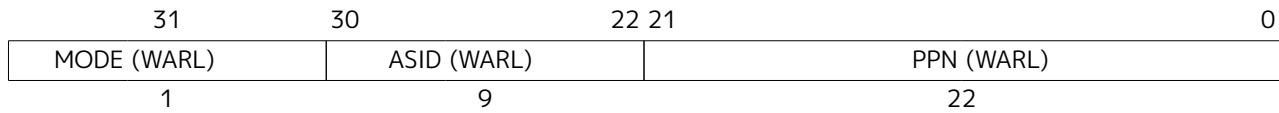
- The hart does not update the **ELP** state; it remains as **NO_LP_EXPECTED**.
- The **LPAD** instruction operates as a no-op.

The Zicfiss extension adds the **SSE** field in **senvcfg**. When the **SSE** field is set to 1, the Zicfiss extension is activated in VU/U-mode. When the **SSE** field is 0, the Zicfiss extension remains inactive in VU/U-mode, and the following rules apply:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- When **menvcfg.SSE** is one, **SSAMOSWAP.W/D** raises an illegal-instruction exception in U-mode and a virtual-instruction exception in VU-mode.

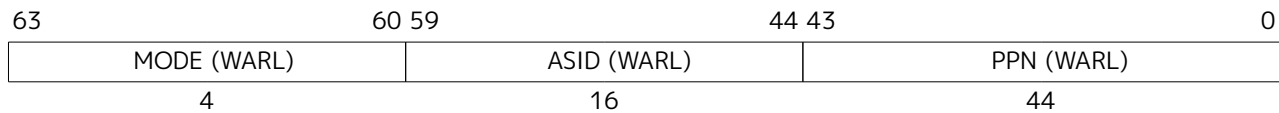
4.1.11. Supervisor Address Translation and Protection (**satp**) Register

The **satp** CSR is an SXLEN-bit read/write register, formatted as shown in [Figure 56](#) for SXLEN=32 and [Figure 57](#) for SXLEN=64, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the **MODE** field, which selects the current address-translation scheme. Further details on the access to this register are described in [Section 3.1.6.6](#).

Figure 56. Supervisor address translation and protection (**satp**) register when SXLEN=32.

Storing a PPN in **satp**, rather than a physical address, supports a physical address space larger than 4 GiB for RV32.

The **satp.PPN** field might not be capable of holding all physical page numbers. Some platform standards might place constraints on the values **satp.PPN** may assume, e.g., by requiring that all physical page numbers corresponding to main memory be representable.

Figure 57. Supervisor address translation and protection (**satp**) register when SXLEN=64, for MODE values Bare, Sv39, Sv48, and Sv57.

We store the ASID and the page table base address in the same CSR to allow the pair to be changed atomically on a context switch. Swapping them non-atomically could pollute the old virtual address space with new translations, or vice-versa. This approach also slightly reduces the cost of a context switch.

Table 125 shows the encodings of the MODE field when SXLEN=32 and SXLEN=64. When MODE=Bare, supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme described in Section 3.7. To select MODE=Bare, software must write zero to the remaining fields of **satp** (bits 30–0 when SXLEN=32, or bits 59–0 when SXLEN=64). Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on address translation and protection behavior.

When SXLEN=32, the **satp** encodings corresponding to MODE=Bare and ASID[8:7]=3 are designated for custom use, whereas the encodings corresponding to MODE=Bare and ASID[8:7]≠3 are reserved for future standard use. When SXLEN=64, all **satp** encodings corresponding to MODE=Bare are reserved for future standard use.



Version 1.11 of this standard stated that the remaining fields in **satp** had no effect when MODE=Bare. Making these fields reserved facilitates future definition of additional translation and protection modes, particularly in RV32, for which all patterns of the existing MODE field have already been allocated.

If an implementation supports the Svbare extension, then the **satp** register's MODE field must be capable of holding the value Bare.

When SXLEN=32, the only other valid setting for MODE is Sv32, a paged virtual-memory scheme described in Section 4.3.

When SXLEN=64, three paged virtual-memory schemes are defined: Sv39, Sv48, and Sv57, described in Section 4.4, Section 4.5, and Section 4.6, respectively. One additional scheme, Sv64, will be defined in a later version of this specification. The remaining MODE settings are reserved for future use and may define different interpretations of the other fields in **satp**.

Implementations are not required to support all MODE settings, and if **satp** is written with an unsupported MODE, the entire write has no effect; no fields in **satp** are modified.

The number of ASID bits is UNSPECIFIED and may be zero. The number of implemented ASID bits, termed *ASIDLEN*, may be determined by writing one to every bit position in the ASID field, then reading back the value in **satp** to see which bit positions in the ASID field hold a one. The least-significant bits of ASID are implemented first: that is, if $ASIDLEN > 0$, $ASID[ASIDLEN-1:0]$ is writable. The maximal value of *ASIDLEN*, termed *ASIDMAX*, is 9 for Sv32 or 16 for Sv39, Sv48, and Sv57.

DRAFT

Table 125. Encoding of **satp** MODE field.

SXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing (see Section 4.3).
SXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	-	Reserved for standard use
8	Sv39	Page-based 39-bit virtual addressing (see Section 4.4).
9	Sv48	Page-based 48-bit virtual addressing (see Section 4.5).
10	Sv57	Page-based 57-bit virtual addressing (see Section 4.6).
11	Sv64	Reserved for page-based 64-bit virtual addressing.
12-13	-	Reserved for standard use
14-15	-	Designated for custom use

For many applications, the choice of page size has a substantial performance impact. A large page size increases TLB reach and loosens the associativity constraints on virtually indexed, physically tagged caches. At the same time, large pages exacerbate internal fragmentation, wasting physical memory and possibly cache capacity.

After much deliberation, we have settled on a conventional page size of 4 KiB for both RV32 and RV64. We expect this decision to ease the porting of low-level runtime software and device drivers. The TLB reach problem is ameliorated by transparent superpage support in modern operating systems. ([Navarro et al., 2002](#)) Additionally, multi-level TLB hierarchies are quite inexpensive relative to the multi-level cache hierarchies whose address space they map.

The **satp** CSR is considered active when the effective privilege mode is S-mode or U-mode. Executions of the address-translation algorithm may only begin using a given value of **satp** when **satp** is active.

Translations that began while **satp** was active are not required to complete or terminate when **satp** is no longer active, unless an **SFENCE.VMA** instruction matching the address and ASID is executed. The **SFENCE.VMA** instruction must be used to ensure that updates to the address-translation data structures are observed by subsequent implicit reads to those structures by a hart.

Note that writing **satp** does not imply any ordering constraints between page-table updates and subsequent address translations, nor does it imply any invalidation of address-translation caches. If the new address space's page tables have been modified, or if an ASID is reused, it may be necessary to execute an **SFENCE.VMA** instruction (see [Section 4.2.1](#)) after, or in some cases before, writing **satp**.

Not imposing upon implementations to flush address-translation caches upon **satp** writes reduces the cost of context switches, provided a sufficiently large ASID space.

4.1.12. Supervisor Timer (**stimecmp**) Register

The **stimecmp** CSR is a 64-bit register and has 64-bit precision on all RV32 and RV64 systems. In RV32 only, accesses to the **stimecmp** CSR access the low 32 bits, while accesses to the **stimecmph** CSR access the high 32 bits of **stimecmp**.

A supervisor timer interrupt becomes pending, as reflected in the STIP bit in the **mip** and **sip** registers whenever **time** contains a value greater than or equal to **stimecmp**, treating the values as unsigned integers.

If the result of this comparison changes, it is guaranteed to be reflected in STIP eventually, but not necessarily immediately. The interrupt remains posted until `stimecmp` becomes greater than `time`, typically as a result of writing `stimecmp`. The interrupt will be taken based on the standard interrupt enable and delegation rules.



A spurious timer interrupt might occur if an interrupt handler advances `stimecmp` then immediately returns, because STIP might not yet have fallen in the interim. All software should be written to assume this event is possible, but most software should assume this event is extremely unlikely. It is almost always more performant to incur an occasional spurious timer interrupt than to poll STIP until it falls.

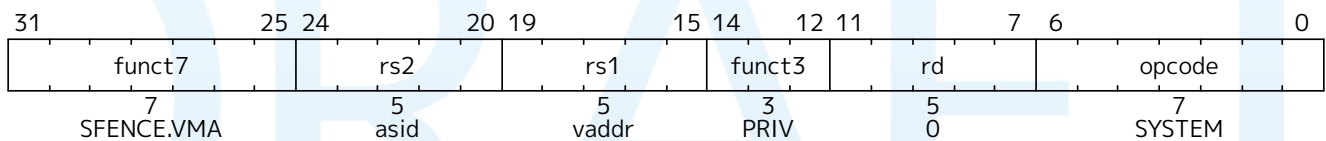


In systems in which a supervisor execution environment (SEE) provides timer facilities via an SBI function call, this SBI call will continue to support requests to schedule a timer interrupt. The SEE will simply make use of `stimecmp`, changing its value as appropriate. This ensures compatibility with existing S-mode software that uses this SEE facility, while new S-mode software takes advantage of `stimecmp` directly.)

4.2. Supervisor Instructions

In addition to the SRET instruction defined in [Section 3.3.2](#), one new supervisor-level instruction is provided.

4.2.1. Supervisor Memory-Management Fence Instruction



The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by `rs1` and `rs2`, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see [Section 4.3.2](#)). Further details on the behavior of this instruction are described in [Section 3.1.6.6](#) and [Section 3.7.2](#).



The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

SFENCE.VMA orders only the local hart's implicit references to the memory-management data structures.



Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VMA in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shutdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *rs2* can specify the address space. The behavior of SFENCE.VMA depends on *rs1* and *rs2* as follows:

- If *rs1*=**x0** and *rs2*=**x0**, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address spaces.
- If *rs1*=**x0** and *rs2*≠**x0**, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *rs2*. Accesses to *global* mappings (see [Section 4.3.1](#)) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register *rs2*, except for entries containing global mappings.
- If *rs1*≠**x0** and *rs2*=**x0**, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces.
- If *rs1*≠**x0** and *rs2*≠**x0**, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for the address space identified by integer register *rs2*. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1* and that match the address space identified by integer register *rs2*, except for entries containing global mappings.

If the value held in *rs1* is not a valid virtual address, then the SFENCE.VMA instruction has no effect. No exception is raised in this case.



*It is always legal to over-fence, e.g., by fencing only based on a subset of the bits in *rs1* and/or *rs2*, and/or by simply treating all SFENCE.VMA instructions as having *rs1*=**x0** and/or *rs2*=**x0**. For example, simpler implementations can ignore the virtual address in *rs1* and the ASID value in *rs2* and always perform a global fence. The choice not to raise an exception when an invalid virtual address is held in *rs1* facilitates this type of simplification.*

When *rs2*≠**x0**, bits SXLEN-1:ASIDMAX of the value held in *rs2* are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN<ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *rs2*.

An implicit read of the memory-management data structures may return any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. The ordering implied by SFENCE.VMA does not place implicit reads and writes to the memory-management data structures into the global memory order in a way that interacts cleanly with the standard RVWMO ordering rules. In particular, even though an SFENCE.VMA orders prior explicit accesses before subsequent implicit accesses, and those implicit accesses are ordered before their associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.



A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address.

For example, if a leaf PTE is modified and the corresponding virtual address is accessed without a subsuming SFENCE.VMA having been executed in between, then either the new translation or any older translation since the last subsuming SFENCE.VMA was executed will

be used. It is unpredictable which translation will be chosen from that set, and subsequent accesses to the same virtual address might use different translations from that set. But the behavior of such accesses is otherwise well defined.

This property applies even if the virtual-address width for that translation differs from the width currently specified by **satp.MODE**. For a given virtual address and ASID, any translation since the last subsuming **SFENCE.VMA** might be used, even if that translation used a virtual address of a different width. Similarly, for a given virtual address, any global translation since the last subsuming **SFENCE.VMA** might be used, regardless of both ASID and virtual-address width.

In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an **SFENCE.VMA** with **rs1=x0**. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the **satp** register or a subsequent valid (V=1) translation data structure entry, and must only raise exceptions for implicit accesses that are generated as a result of instruction execution, not those that are performed speculatively.

Changes to the **sstatus** fields SUM and MXR take effect immediately, without the need to execute an **SFENCE.VMA** instruction. Changing **satp.MODE** from Bare to other modes and vice versa also takes effect immediately, without the need to execute an **SFENCE.VMA** instruction. Likewise, changes to **satp.ASID** take effect immediately.

The following common situations typically require executing an **SFENCE.VMA** instruction:



- When software recycles an ASID (i.e., reassociates it with a different page table), it should first change **satp** to point to the new page table using the recycled ASID, then execute **SFENCE.VMA** with **rs1=x0** and **rs2** set to the recycled ASID. Alternatively, software can execute the same **SFENCE.VMA** instruction while a different ASID is loaded into **satp**, provided the next time **satp** is loaded with the recycled ASID, it is simultaneously loaded with the new page table.
- If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every **satp** write, software should execute **SFENCE.VMA** with **rs1=x0**. In the common case that no global translations have been modified, **rs2** should be set to a register other than **x0** but which contains the value zero, so that global translations are not flushed.
- If software modifies a non-leaf PTE, it should execute **SFENCE.VMA** with **rs1=x0**. If any PTE along the traversal path had its G bit set, **rs2** must be **x0**; otherwise, **rs2** should be set to the ASID for which the translation is being modified.
- If software modifies a leaf PTE, it should execute **SFENCE.VMA** with **rs1** set to a virtual

address within the page. If any PTE along the traversal path had its *G* bit set, *rs2* must be *x0*; otherwise, *rs2* should be set to the ASID for which the translation is being modified.

- For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the *SFENCE.VMA* lazily. After modifying the PTE but before executing *SFENCE.VMA*, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute *SFENCE.VMA* in accordance with the previous bullet point.

If a hart employs an address-translation cache, that cache must appear to be private to that hart. In particular, the meaning of an ASID is local to a hart; software may choose to use the same ASID to refer to different address spaces on different harts.



A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shutdown. However, as OSes have evolved to significantly reduce the scope of TLB shutdowns using novel ASID-management techniques, we expect the local-ASID scheme to remain attractive for its simplicity and possibly better scalability.

For implementations that make *satp.MODE* read-only zero (always Bare), attempts to execute an *SFENCE.VMA* instruction might raise an illegal-instruction exception.

No *SFENCE.VMA* is required after enabling or disabling pointer masking (see [Section 6.10](#)), as pointer masking applies to the effective address only and does not affect any memory-management data structures.

4.3. Sv32: Page-Based 32-bit Virtual-Memory Systems

When Sv32 is written to the *MODE* field in the *satp* register (see [Section 4.1.11](#)), the supervisor operates in a 32-bit paged virtual-memory system. In this mode, supervisor and user virtual addresses are translated into supervisor physical addresses by traversing a radix-tree page table. Sv32 is supported when *SXLEN*=32 and is designed to include mechanisms sufficient for supporting modern Unix-based operating systems.



The initial RISC-V paged virtual-memory architectures have been designed as straightforward implementations to support existing operating systems. We have architected page table layouts to support a hardware page-table walker. Software TLB refills are a performance bottleneck on high-performance systems, and are especially troublesome with decoupled coprocessors. An implementation can choose to implement software TLB refills using a machine-mode trap handler as an extension to M-mode.

Some ISAs architecturally expose virtually indexed, physically tagged caches, in that accesses to the same physical address via different virtual addresses might not be coherent unless the virtual addresses lie within the same cache set. Implicitly, this specification does not permit such behavior to be architecturally exposed.

4.3.1. Addressing and Memory Protection

Sv32 implementations support a 32-bit virtual address space, divided into pages. An Sv32 virtual address is partitioned into a virtual page number (VPN) and page offset, as shown in [Figure 58](#). When Sv32 virtual memory mode is selected in the *MODE* field of the *satp* register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12-bit page offset is untranslated. The resulting supervisor-level

physical addresses are then checked using any physical memory protection structures ([Section 3.7](#)), before being directly converted to machine-level physical addresses. If necessary, supervisor-level physical addresses are zero-extended to the number of physical address bits found in the implementation.



For example, consider an RV32 system supporting 34 bits of physical address. When the value of `satp.MODE` is Sv32, a 34-bit physical address is produced directly, and therefore no zero extension is needed. When the value of `satp.MODE` is Bare, the 32-bit virtual address is translated (unmodified) into a 32-bit physical address, and then that physical address is zero-extended into a 34-bit machine-level physical address.

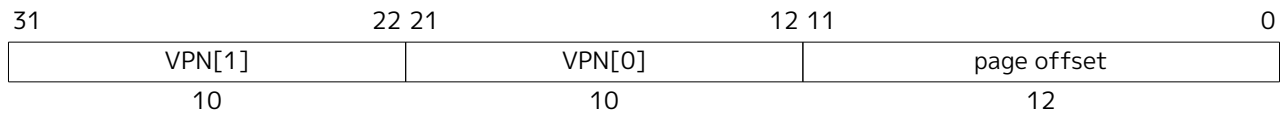


Figure 58. Sv32 virtual address.

Sv32 page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the **satp** register.

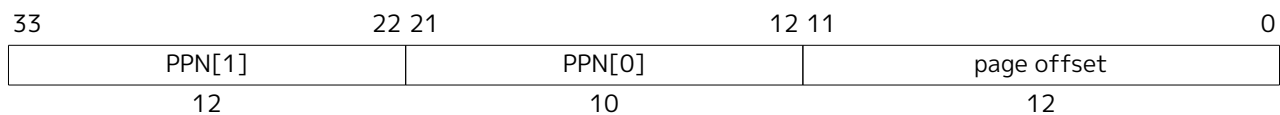


Figure 59. SV32 physical address.

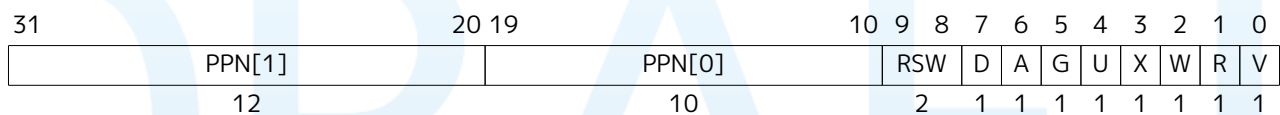


Figure 60. Sv32 page table entry.

The PTE format for Sv32 is shown in [Figure 60](#). The V bit indicates whether the PTE is valid; if it is 0, all other bits in the PTE are don't-cares and may be used freely by software. The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE. Writable pages must also be marked readable; the contrary combinations are reserved for future use. [Table 126](#) summarizes the encoding of the permission bits.

Table 126. Encoding of PTE R/W/X fields.

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	<i>Reserved for future use.</i>
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	<i>Reserved for future use.</i>
1	1	1	Read-write-execute page.

Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load, load-reserved, or cache-block management instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional, AMO, or cache-block zero instruction instruction whose effective address lies within a page without write permissions raises a store page-fault exception.



AMOs never raise load page-fault exceptions. Since any unreadable page is also unwritable, attempting to perform an AMO on an unreadable page always raises a store page-fault exception.

The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the `sstatus` register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which case, supervisor code will fault on accesses to user-mode pages. Irrespective of SUM, the supervisor may not execute code on pages with U=1.



An alternative PTE format would support different permissions for supervisor and user. We omitted this feature because it would be largely redundant with the SUM mechanism (see [Section 4.1.1.2](#)) and would require more encoding space in the PTE.

The G bit designates a *global* mapping. Global mappings are those that exist in all address spaces. For non-leaf PTEs, the global setting implies that all mappings in the subsequent levels of the page table are global. Note that failing to mark a global mapping as global merely reduces performance, whereas marking a non-global mapping as global is a software bug that, after switching to an address space with a different non-global mapping for that address range, can unpredictably result in either mapping being used.



Global mappings need not be stored redundantly in address-translation caches for multiple ASIDs. Additionally, they need not be flushed from local address-translation caches when an `SFENCE.VMA` instruction is executed with `rs2≠x0`.

The RSW field is reserved for use by supervisor software; the implementation shall ignore this field.

Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared.

Two schemes to manage the A and D bits are defined:

- The *Svade* extension: when a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When the *Svade* extension is not implemented, the following scheme applies.

When a virtual page is accessed and the A bit is clear, the PTE is updated to set the A bit. When the virtual page is written and the D bit is clear, the PTE is updated to set the D bit. When G-stage address translation is in use and is not Bare, the G-stage virtual pages may be accessed or written by implicit accesses to VS-level memory management data structures, such as page tables.

When two-stage address translation is in use, an explicit access may cause both VS-stage and G-stage PTEs to be updated. The following rules apply to all PTE updates caused by an explicit or an implicit memory accesses.

The PTE update must be atomic with respect to other accesses to the PTE, and must atomically perform all page-table walk checks for that leaf PTE as part of, and before, conditionally updating the PTE value. Updates of the A bit may be performed as a result of speculation, even if the associated memory access ultimately is not performed architecturally. However, updates to the D bit, resulting from an explicit store, must be exact (i.e., non-speculative), and observed in program order by the local hart. When two-stage address translation is active, updates to the D bit in G-stage PTEs may be performed by an implicit access to a VS-stage PTE, if the G-stage PTE provides write permission, before any speculative access to the VS-stage PTE.

The PTE update must appear in the global memory order before the memory access that caused the PTE update and before any subsequent explicit memory access to that virtual page by the local hart. The ordering on loads and stores provided by FENCE instructions and the acquire/release bits on atomic instructions also orders the PTE updates associated with those loads and stores as observed by remote harts.

The PTE update is not required to be atomic with respect to the memory access that caused the update and a trap may occur between the PTE update and the memory access that caused the PTE update. If a trap occurs then the A and/or D bit may be updated but the memory access that caused the PTE update might not occur. The hart must not perform the memory access that caused the PTE update before the PTE update is globally visible.

The page tables must be located in memory with hardware page-table write access and *RsrvEventual* PMA.

All harts in a system must employ the same PTE-update scheme as each other.

The PTE updates due to memory accesses ordered-after a FENCE are not themselves ordered by the FENCE.

Simpler implementations may order the Page Table Entry (PTE) update to precede all subsequent explicit memory accesses, as opposed to ensuring that the PTE update is precisely sequenced before subsequent explicit memory accesses to the associated virtual page.

Prior versions of this specification required PTE A bit updates to be exact, but allowing the A bit to be updated as a result of speculation simplifies the implementation of address translation prefetchers. System software typically uses the A bit as a page replacement policy hint, but does not require exactness for functional correctness. On the other hand, D bit updates are still required to be exact and performed in program order, as the D bit affects the functional correctness of page eviction.

Implementations are of course still permitted to perform both A and D bit updates only in an exact manner.

In both cases, requiring atomicity ensures that the PTE update will not be interrupted by other intervening writes to the page table, as such interruptions could lead to A/D bits being set on PTEs that have been reused for other purposes, on memory that has been reclaimed for other purposes, and so on. Simple implementations may instead generate page-fault exceptions.

The A and D bits are never cleared by the implementation. If the supervisor software does not rely on accessed and/or dirty bits, e.g. if it does not swap memory pages to secondary storage or if the pages are being used to map I/O space, it should always set them to 1 in the PTE to improve performance.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB *megapages*. A megapage must be virtually and physically aligned to a 4 MiB boundary; a page-fault exception is raised if the physical address is insufficiently aligned.

For non-leaf PTEs, the D, A, and U bits are reserved for future standard use. Until their use is defined by a standard extension, they must be cleared by software for forward compatibility.

For implementations with both page-based virtual memory and the "A" standard extension, the LR/SC reservation set must lie completely within a single base physical page (i.e., a naturally aligned 4 KiB

physical-memory region).

On some implementations, misaligned loads, stores, and instruction fetches may also be decomposed into multiple accesses, some of which may succeed before a page-fault exception occurs. In particular, a portion of a misaligned store that passes the exception check may become visible, even if another portion fails the exception check. The same behavior may manifest for stores wider than XLEN bits (e.g., the FSD instruction in RV32D), even when the store address is naturally aligned.

4.3.2. Virtual Address Translation Process

A virtual address va is translated into a physical address pa as follows:

1. Let a be $\text{satp.ppn} \times \text{PAGESIZE}$, and let $i = \text{LEVELS} - 1$. (For Sv32, $\text{PAGESIZE} = 2^{12}$ and $\text{LEVELS} = 2$.) The **satp** register must be *active*, i.e., the effective privilege mode must be S-mode or U-mode.
2. Let pte be the value of the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$. (For Sv32, $\text{PTESIZE} = 4$.) If accessing pte violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
3. If $pte.v = 0$, or if $pte.r = 0$ and $pte.w = 1$, or if any bits or encodings that are reserved for future standard use are set within pte , stop and raise a page-fault exception corresponding to the original access type.
4. Otherwise, the PTE is valid. If $pte.r = 1$ or $pte.x = 1$, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let $i = i - 1$. If $i < 0$, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let $a = pte.ppn \times \text{PAGESIZE}$ and go to step 2.
5. A leaf PTE has been reached. If $i > 0$ and $pte.ppn[i-1:0] \neq 0$, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.
6. Determine if the requested memory access is allowed by the $pte.u$ bit, given the current privilege mode and the value of the SUM and MXR fields of the **mstatus** register. If not, stop and raise a page-fault exception corresponding to the original access type.
7. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, and $pte.x$ bits, given the Shadow Stack Memory Protection rules. If not, stop and raise an access-fault exception.
8. Determine if the requested memory access is allowed by the $pte.r$, $pte.w$, and $pte.x$ bits. If not, stop and raise a page-fault exception corresponding to the original access type.
9. If $pte.a = 0$, or if the original memory access is a store and $pte.d = 0$:
 - If the Svade extension is implemented, stop and raise a page-fault exception corresponding to the original access type.
 - If a store to the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$ would violate a PMA or PMP check, raise an access-fault exception corresponding to the original access type.
 - Perform the following steps atomically:
 - Compare pte to the value of the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$.
 - If the values match, set $pte.a$ to 1 and, if the original memory access is a store, also set $pte.d$ to 1. Then store pte to the PTE at address $a + va.vpn[i] \times \text{PTESIZE}$.
 - If the comparison fails, return to step 2.
10. The translation is successful. The translated physical address is given as follows:
 - $pa.pgoff = va.pgoff$.
 - If $i > 0$, then this is a superpage translation and $pa.ppn[i-1:0] = va.vpn[i-1:0]$.
 - $pa.ppn[\text{LEVELS}-1:i] = pte.ppn[\text{LEVELS}-1:i]$.

All implicit accesses to the address-translation data structures in this algorithm are performed using width

PTESIZE.



This implies, for example, that an Sv48 implementation may not use two separate 4 B reads to non-atomically access a single 8 B PTE, and that A/D bit updates performed by the implementation are treated as atomically updating the entire PTE, rather than just the A and/or D bit alone (even though the PTE value does not otherwise change).

The results of implicit address-translation reads in step 2 may be held in a read-only, incoherent *address-translation cache* but not shared with other harts. The address-translation cache may hold an arbitrary number of entries, including an arbitrary number of entries for the same address and ASID. Entries in the address-translation cache may then satisfy subsequent step 2 reads if the ASID associated with the entry matches the ASID loaded in step 0 or if the entry is associated with a *global* mapping. To ensure that implicit reads observe writes to the same memory locations, an SFENCE.VMA instruction must be executed after the writes to flush the relevant cached translations.

The address-translation cache cannot be used in step 9; accessed and dirty bits may only be updated in memory directly.



It is permitted for multiple address-translation cache entries to co-exist for the same address. This represents the fact that in a conventional TLB hierarchy, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with $rs1=x0$, or if multiple TLBs exist in parallel at a given level of the hierarchy. In this case, just as if an SFENCE.VMA is not executed between a write to the memory-management tables and subsequent implicit read of the same address: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.

Implementations may also execute the address-translation algorithm speculatively at any time, for any virtual address, as long as **satp** is active (as defined in [Section 4.1.11](#)). Such speculative executions have the effect of pre-populating the address-translation cache.

Speculative executions of the address-translation algorithm behave as non-speculative executions of the algorithm do, except that they must not set the dirty bit for a PTE, they must not trigger an exception, and they must not create address-translation cache entries if those entries would have been invalidated by any SFENCE.VMA instruction executed by the hart since the speculative execution of the algorithm began.

For instance, it is illegal for both non-speculative and speculative executions of the translation algorithm to begin, read the level 2 page table, pause while the hart executes an SFENCE.VMA with $rs1=rs2=x0$, then resume using the now-stale level 2 PTE, as subsequent implicit reads could populate the address-translation cache with stale PTEs.



In many implementations, an SFENCE.VMA instruction with $rs1=x0$ will therefore either terminate all previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable), or simply wait for them to complete (in which case any address-translation cache entries created will be invalidated by the SFENCE.VMA as appropriate). Likewise, an SFENCE.VMA instruction with $rs1 \neq x0$ generally must either ensure that previously-launched speculative executions of the address-translation algorithm (for the specified ASID, if applicable) are prevented from creating new address-translation cache entries mapping leaf PTEs, or wait for them to complete.

A consequence of implementations being permitted to read the translation data structures arbitrarily early and speculatively is that at any time, all page table entries reachable by executing the algorithm may be loaded into the address-translation cache.

Although it would be uncommon to place page tables in non-idempotent memory, there is no

explicit prohibition against doing so. Since the algorithm may only touch page tables reachable from the root page table indicated in **satp**, the range of addresses that an implementation's page-table walker will touch is fully under supervisor control.

The algorithm does not admit the possibility of ignoring high-order PPN bits for implementations with narrower physical addresses.

4.4. Sv39: Page-Based 39-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 39-bit virtual address spaces. The design of Sv39 follows the overall scheme of Sv32, and this section details only the differences between the schemes.



We specified multiple virtual memory systems for RV64 to relieve the tension between providing a large address space and minimizing address-translation cost. For many systems, 39 bits of virtual-address space is ample, and so Sv39 suffices. Sv48 increases the virtual address space to 48 bits, but increases the physical memory capacity dedicated to page tables, the latency of page-table traversals, and the size of hardware structures that store virtual addresses. Sv57 increases the virtual address space, page table capacity requirement, and translation latency even further.

4.4.1. Addressing and Memory Protection

Sv39 implementations support a 39-bit virtual address space, divided into pages. An Sv39 address is partitioned as shown in Figure 61. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–39 all equal to bit 38, or else a page-fault exception will occur. The 27-bit VPN is translated into a 44-bit PPN via a three-level page table, while the 12-bit page offset is untranslated.



When mapping between narrower and wider addresses, RISC-V zero-extends a narrower physical address to a wider size. The mapping between 64-bit virtual addresses and the 39-bit usable address space of Sv39 is not based on zero extension but instead follows an entrenched convention that allows an OS to use one or a few of the most-significant bits of a full-size (64-bit) virtual address to quickly distinguish user and supervisor address regions.

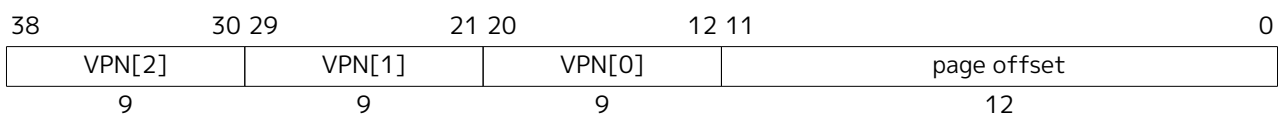


Figure 61. Sv39 virtual address.

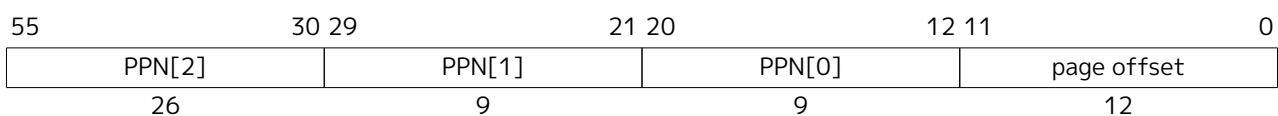


Figure 62. Sv39 physical address.

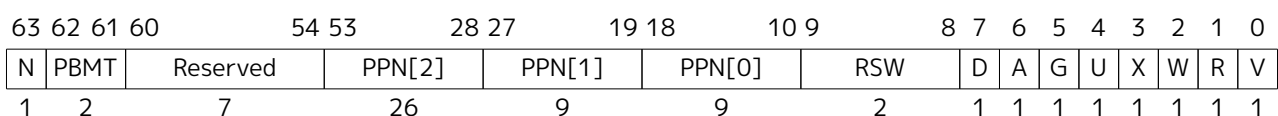


Figure 63. Sv39 page table entry.

Sv39 page tables contain 2^9 page table entries (PTEs), eight bytes each. A page table is exactly the size of a

page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the **satp** register's PPN field.

The PTE format for Sv39 is shown in [Figure 63](#). Bits 9-0 have the same meaning as for Sv32. Bit 63 is reserved for use by the Svn timer extension in [Section 7.1](#). If Svn timer is not implemented, bit 63 remains reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 62-61 are reserved for use by the Svpbmt extension in [Section 7.2](#). If Svpbmt is not implemented, bits 62-61 remain reserved and must be zeroed by software for forward compatibility, or else a page-fault exception is raised. Bits 60-54 are reserved for future standard use and, until their use is defined by some standard extension, must be zeroed by software for forward compatibility. If any of these bits are set, a page-fault exception is raised.



We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency. These reserved bits may also be used to facilitate research experimentation. The cost is reducing the physical address space, but is presently ample. When it no longer suffices, the reserved bits that remain unallocated could be used to expand the physical address space.

Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv39 supports 2 MiB megapages and 1 GiB gigapages, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in [Section 4.3.2](#), except LEVELS equals 3 and PTESIZE equals 8.

4.5. Sv48: Page-Based 48-bit Virtual-Memory System

This section describes a simple paged virtual-memory system for SXLEN=64, which supports 48-bit virtual address spaces. Sv48 is intended for systems for which a 39-bit virtual address space is insufficient. It closely follows the design of Sv39, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv48 must also support Sv39.



Systems that support Sv48 can also support Sv39 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv39.

4.5.1. Addressing and Memory Protection

Sv48 implementations support a 48-bit virtual address space, divided into pages. An Sv48 address is partitioned as shown in [Figure 64](#). Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur. The 36-bit VPN is translated into a 44-bit PPN via a four-level page table, while the 12-bit page offset is untranslated.

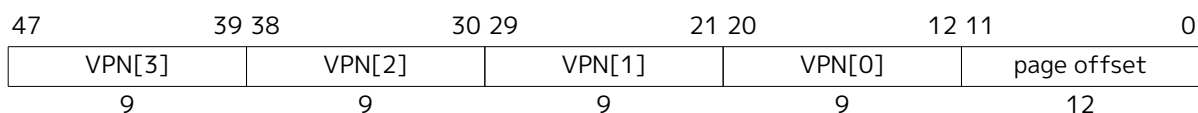


Figure 64. Sv48 virtual address.

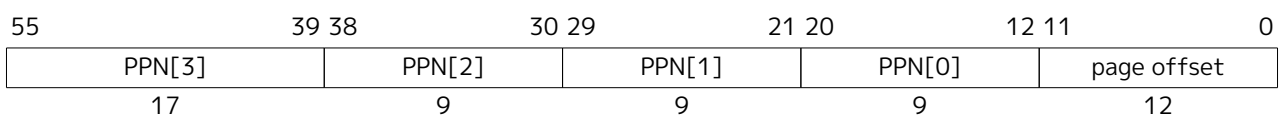


Figure 65. Sv48 physical address.

63	62	61	60	54	53	37	36	28	27	19	18	10	9	8	7	6	5	4	3	2	1	0
N	PBMT	Reserved	PPN[3]	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V							
1	2	7	17	9	9	9	2	1	1	1	1	1	1	1	1							

Figure 66. Sv48 page table entry.

The PTE format for Sv48 is shown in Figure 66. Bits 63–54 and 9–0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv48 supports 2 MiB megapages, 1 GiB gigapages, and 512 GiB terapages, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in Section 4.3.2, except LEVELS equals 4 and PTESIZE equals 8.

4.6. Sv57: Page-Based 57-bit Virtual-Memory System

This section describes a simple paged virtual-memory system designed for RV64 systems, which supports 57-bit virtual address spaces. Sv57 is intended for systems for which a 48-bit virtual address space is insufficient. It closely follows the design of Sv48, simply adding an additional level of page table, and so this chapter only details the differences between the two schemes.

Implementations that support Sv57 must also support Sv48.



Systems that support Sv57 can also support Sv48 at essentially no cost, and so should do so to maintain compatibility with supervisor software that assumes Sv48.

4.6.1. Addressing and Memory Protection

Sv57 implementations support a 57-bit virtual address space, divided into pages. An Sv57 address is partitioned as shown in Figure 67. Instruction fetch addresses and load and store effective addresses, which are 64 bits, must have bits 63–57 all equal to bit 56, or else a page-fault exception will occur. The 45-bit VPN is translated into a 44-bit PPN via a five-level page table, while the 12-bit page offset is untranslated.

56	48	47	39	38	30	29	21	20	12	11	0
VPN[4]		VPN[3]		VPN[2]		VPN[1]		VPN[0]		page offset	
9		9		9		9		9		12	

Figure 67. Sv57 virtual address.

55	48	47	39	38	30	29	21	20	12	11	0
PPN[4]		PPN[3]		PPN[2]		PPN[1]		PPN[0]		page offset	
8		9		9		9		9		12	

Figure 68. Sv57 physical address.

63	62	61	60	54	53	10	9	8	7	6	5	4	3	2	1	0
N	PBMT	Reserved			PPN		RSW		D	A	G	U	X	W	R	V
1	2	7			44		2		1	1	1	1	1	1	1	1

53	46	45	37	36	28	27	19	18	10
PPN[3]		PPN[2]		PPN[1]		PPN[0]			
8		9		9		9			

Figure 69. Sv57 page table entry.

The PTE format for Sv57 is shown in [Figure 69](#). Bits 63–54 and 9–0 have the same meaning as for Sv39. Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv57 supports 2 MiB megapages, 1 GiB gigapages, 512 GiB terapages, and 256 TiB petapages, each of which must be virtually and physically aligned to a boundary equal to its size. A page-fault exception is raised if the physical address is insufficiently aligned.

The algorithm for virtual-to-physical address translation is the same as in [Section 4.3.2](#), except LEVELS equals 5 and PTESIZE equals 8.

DRAFT

Chapter 5. "H" Extension for Hypervisor Support, Version 1.0

This chapter describes the RISC-V hypervisor extension, which virtualizes the supervisor-level architecture to support the efficient hosting of guest operating systems atop a type-1 or type-2 hypervisor. The hypervisor extension changes supervisor mode into *hypervisor-extended supervisor mode* (HS-mode, or *hypervisor mode* for short), where a hypervisor or a hosting-capable operating system runs. The hypervisor extension also adds another stage of address translation, from *guest physical addresses* to supervisor physical addresses, to virtualize the memory and memory-mapped I/O subsystems for a guest operating system. HS-mode acts the same as S-mode, but with additional instructions and CSRs that control the new stage of address translation and support hosting a guest OS in virtual S-mode (VS-mode). Regular S-mode operating systems can execute without modification either in HS-mode or as VS-mode guests.

In HS-mode, an OS or hypervisor interacts with the machine through the same SBI as an OS normally does from S-mode. An HS-mode hypervisor is expected to implement the SBI for its VS-mode guest.

The hypervisor extension depends on an "I" base integer ISA with 32 **x** registers (RV32I or RV64I), not RV32E or RV64E, which have only 16 **x** registers. CSR **mtval** must not be read-only zero, and standard page-based address translation must be supported, either Sv32 for RV32, or a minimum of Sv39 for RV64.

The hypervisor extension is enabled by setting bit 7 in the **misal** CSR, which corresponds to the letter H. RISC-V harts that implement the hypervisor extension are encouraged not to hardwire **misal**[7], so that the extension may be disabled.



The baseline privileged architecture is designed to simplify the use of classic virtualization techniques, where a guest OS is run at user-level, as the few privileged instructions can be easily detected and trapped. The hypervisor extension improves virtualization performance by reducing the frequency of these traps.

The hypervisor extension has been designed to be efficiently emulable on platforms that do not implement the extension, by running the hypervisor in S-mode and trapping into M-mode for hypervisor CSR accesses and to maintain shadow page tables. The majority of CSR accesses for type-2 hypervisors are valid S-mode accesses so need not be trapped. Hypervisors can support nested virtualization analogously.

5.1. Privilege Modes

The current *virtualization mode*, denoted **V**, indicates whether the hart is currently executing in a guest. When **V**=1, the hart is either in virtual S-mode (VS-mode), or in virtual U-mode (VU-mode) atop a guest OS running in VS-mode. When **V**=0, the hart is either in M-mode, in HS-mode, or in U-mode atop an OS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active (**V**=1) or inactive (**V**=0). [Table 127](#) lists the possible privilege modes of a RISC-V hart with the hypervisor extension.

Table 127. Privilege modes with the hypervisor extension.

Virtualization Mode (V)	Nominal Privilege	Abbreviation	Name	Two-Stage Translation
0	U	U-mode	User mode	Off
0	S	HS-mode	Hypervisor-extended supervisor mode	Off
0	M	M-mode	Machine mode	Off
1	U	VU-mode	Virtual user mode	On
1	S	VS-mode	Virtual supervisor mode	On

For privilege modes U and VU, the *nominal privilege mode* is U, and for privilege modes HS and VS, the nominal privilege mode is S.

HS-mode is more privileged than VS-mode, and VS-mode is more privileged than VU-mode. VS-mode interrupts are globally disabled when executing in U-mode.



This description does not consider the possibility of U-mode or VU-mode interrupts and will be revised if an extension for user-level interrupts is adopted.

5.2. Hypervisor and Virtual Supervisor CSRs

An OS or hypervisor running in HS-mode uses the supervisor CSRs to interact with the exception, interrupt, and address-translation subsystems. Additional CSRs are provided to HS-mode, but not to VS-mode, to manage two-stage address translation and to control the behavior of a VS-mode guest: **hstatus**, **hedeleg**, **hideleg**, **hvip**, **hip**, **hie**, **hgeip**, **hgeie**, **henvcfg**, **henvcfgh**, **hcounteren**, **htimedelta**, **htimedeltah**, **htval**, **htinst**, and **hgap**.

Furthermore, several *virtual supervisor* CSRs (VS CSRs) are replicas of the normal supervisor CSRs. For example, **vsstatus** is the VS CSR that duplicates the usual **sstatus** CSR.

When V=1, the VS CSRs substitute for the corresponding supervisor CSRs, taking over all functions of the usual supervisor CSRs except as specified otherwise. Instructions that normally read or modify a supervisor CSR shall instead access the corresponding VS CSR. When V=1, an attempt to read or write a VS CSR directly by its own separate CSR address causes a virtual-instruction exception. (Attempts from U-mode cause an illegal-instruction exception as usual.) The VS CSRs can be accessed as themselves only from M-mode or HS-mode.

While V=1, the normal HS-level supervisor CSRs that are replaced by VS CSRs retain their values but do not affect the behavior of the machine unless specifically documented to do so. Conversely, when V=0, the VS CSRs do not ordinarily affect the behavior of the machine other than being readable and writable by CSR instructions.

Some standard supervisor CSRs (**senvcfg**, **scounteren**, and **scontext**, possibly others) have no matching VS CSR. These supervisor CSRs continue to have their usual function and accessibility even when V=1, except with VS-mode and VU-mode substituting for HS-mode and U-mode. Hypervisor software is expected to manually swap the contents of these registers as needed.



Matching VS CSRs exist only for the supervisor CSRs that must be duplicated, which are mainly those that get automatically written by traps or that impact instruction execution immediately after trap entry and/or right before SRET, when software alone is unable to swap a CSR at exactly the right moment. Currently, most supervisor CSRs fall into this category, but future ones might not.

In this chapter, we use the term *HSXLEN* to refer to the effective XLEN when executing in HS-mode, and *VSXLEN* to refer to the effective XLEN when executing in VS-mode.

5.2.1. Hypervisor Status (*hstatus*) Register

The *hstatus* register is an HSXLEN-bit read/write register formatted as shown in Figure 70 when HSXLEN=32 and Figure 71 when HSXLEN=64. The *hstatus* register provides facilities analogous to the *mstatus* register for tracking and controlling the exception behavior of a VS-mode guest.

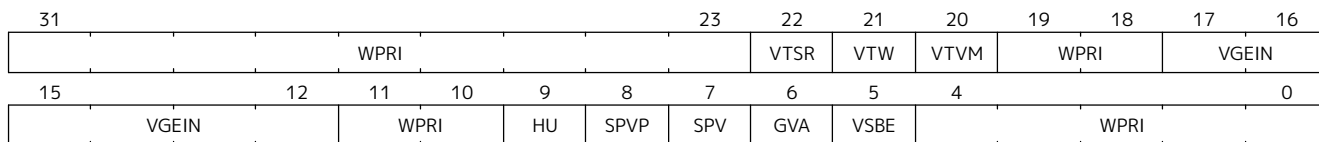


Figure 70. Hypervisor status register (*hstatus*) when HSXLEN=32

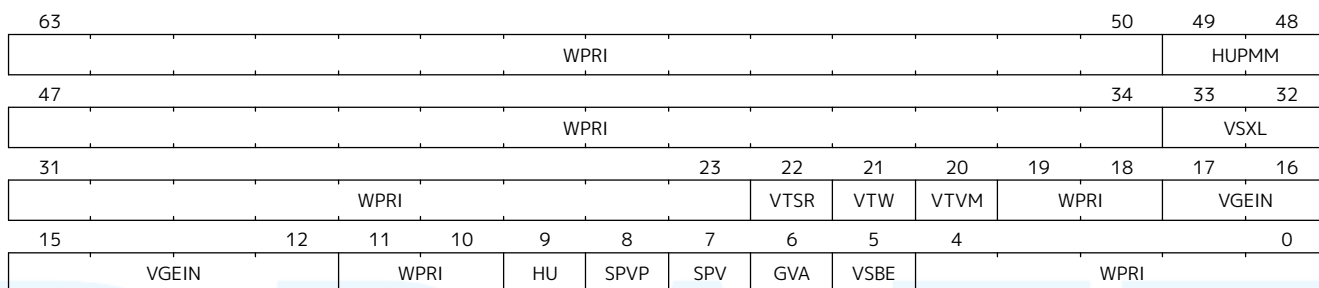


Figure 71. Hypervisor status register (*hstatus*) when HSXLEN=64.

The VSXL field controls the effective XLEN for VS-mode (known as VSXLEN), which may differ from the XLEN for HS-mode (HSXLEN). When HSXLEN=32, the VSXL field does not exist, and VSXLEN=32. When HSXLEN=64, VSXL is a **WARL** field that is encoded the same as the MXL field of *misal*, shown in Table 107. In particular, an implementation may make VSXL be a read-only field whose value always ensures that VSXLEN=HSXLEN.

If HSXLEN is changed from 32 to a wider width, and if field VSXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new HSXLEN.

The *hstatus* fields VTSR, VTW, and VTVM are defined analogously to the *mstatus* fields TSR, TW, and TVM, but affect execution only in VS-mode, and cause virtual-instruction exceptions instead of illegal-instruction exceptions. When VTSR=1, an attempt in VS-mode to execute SRET raises a virtual-instruction exception. When VTW=1 (and assuming *mstatus*.TW=0), an attempt in VS-mode to execute WFI raises a virtual-instruction exception if the WFI does not complete within an implementation-specific, bounded time limit. An implementation may have WFI always raise a virtual-instruction exception in VS-mode when VTW=1 (and *mstatus*.TW=0), even if there are pending globally-disabled interrupts when the instruction is executed. When VTVM=1, an attempt in VS-mode to execute SFENCE.VMA or SINVAL.VMA or to access CSR *satp* raises a virtual-instruction exception.

The VGEIN (Virtual Guest External Interrupt Number) field selects a guest external interrupt source for VS-level external interrupts. VGEIN is a **WLRL** field that must be able to hold values between zero and the maximum guest external interrupt number (known as GEILEN), inclusive. When VGEIN=0, no guest external interrupt source is selected for VS-level external interrupts. GEILEN may be zero, in which case VGEIN may be read-only zero. Guest external interrupts are explained in Section 5.2.4, and the use of VGEIN is covered further in Section 5.2.3.

Field HU (Hypervisor in U-mode) controls whether the virtual-machine load/store instructions, HLV, HLVX, and HSV, can be used also in U-mode. When HU=1, these instructions can be executed in U-mode the same as in HS-mode. When HU=0, all hypervisor instructions cause an illegal-instruction exception in

U-mode.



The HU bit allows a portion of a hypervisor to be run in U-mode for greater protection against software bugs, while still retaining access to a virtual machine's memory.

When Ssnpm extension is implemented, the **HUPMM** field enables or disables pointer masking (see [Section 6.10](#)) for **HLV.*** and **HSV.*** instructions in U-mode, according to the values in [Table 130](#), when their explicit memory access is performed as though in VU-mode. In HS- and M-modes, pointer masking for these instructions is enabled or disabled by **senvcfg.PMM**, when their explicit memory access is performed as though in VU-mode. Setting **henvcfg.PMM** enables or disables pointer masking for **HLV.*** and **HSV.*** when their explicit memory access is performed as though in VS-mode. When the Ssnpm extension is not implemented, the **HUPMM** field is read-only zero. The **HUPMM** field is read-only zero for RV32.



*The hypervisor should copy the value written to **senvcfg.PMM** by the guest to the **hstatus.HUPMM** field prior to invoking **HLV.*** or **HSV.*** instructions in U-mode.*

The SPV bit (Supervisor Previous Virtualization mode) is written by the implementation whenever a trap is taken into HS-mode. Just as the SPP bit in **sstatus** is set to the (nominal) privilege mode at the time of the trap, the SPV bit in **hstatus** is set to the value of the virtualization mode V at the time of the trap. When an SRET instruction is executed when V=0, V is set to SPV.

When V=1 and a trap is taken into HS-mode, bit SPVP (Supervisor Previous Virtual Privilege) is set to the nominal privilege mode at the time of the trap, the same as **sstatus.SPP**. But if V=0 before a trap, SPVP is left unchanged on trap entry. SPVP controls the effective privilege of explicit memory accesses made by the virtual-machine load/store instructions, **HLV**, **HLVX**, and **HSV**.



*Without SPVP, if instructions **HLV**, **HLVX**, and **HSV** looked instead to **sstatus.SPP** for the effective privilege of their memory accesses, then, even with HU=1, U-mode could not access virtual machine memory at VS-level, because to enter U-mode using SRET always leaves SPP=0. Unlike SPP, field SPVP is untouched by transitions back-and-forth between HS-mode and U-mode.*

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into HS-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault) that writes a guest virtual address to **stval**, GVA is set to 1. For any other trap into HS-mode, GVA is set to 0.

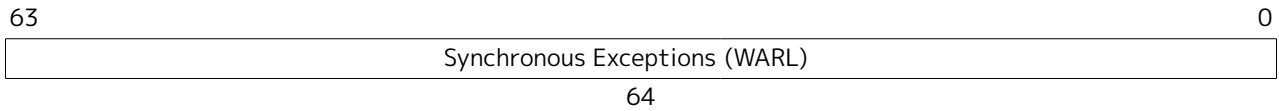
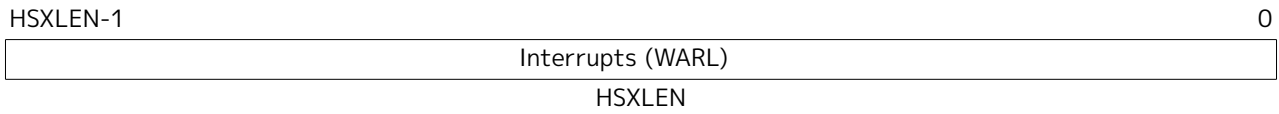


*For breakpoint and memory access traps that write a nonzero value to **stval**, GVA is redundant with field SPV (the two bits are set the same) except when the explicit memory access of an **HLV**, **HLVX**, or **HSV** instruction causes a fault. In that case, SPV=0 but GVA=1.*

The VSBE bit is a **WARL** field that controls the endianness of explicit memory accesses made from VS-mode. If VSBE=0, explicit load and store memory accesses made from VS-mode are little-endian, and if VSBE=1, they are big-endian. VSBE also controls the endianness of all implicit accesses to VS-level memory management data structures, such as page tables. An implementation may make VSBE a read-only field that always specifies the same endianness as HS-mode.

5.2.2. Hypervisor Trap Delegation (**hedeleg** and **hideleg**) Registers

Register **hedeleg** is a 64-bit read/write register, formatted as shown in [Figure 72](#). Register **hideleg** is an HSXLEN-bit read/write register, formatted as shown in [Figure 73](#). By default, all traps at any privilege level are handled in M-mode, though M-mode usually uses the **medeleg** and **mideleg** CSRs to delegate some traps to HS-mode. The **hedeleg** and **hideleg** CSRs allow these traps to be further delegated to a VS-mode guest; their layout is the same as **medeleg** and **mideleg**.

Figure 72. Hypervisor exception delegation register (**hedeleg**).Figure 73. Hypervisor interrupt delegation register (**hideleg**).

A synchronous trap that has been delegated to HS-mode (using **medeleg**) is further delegated to VS-mode if $V=1$ before the trap and the corresponding **hedeleg** bit is set. Each bit of **hedeleg** shall be either writable or read-only zero. Many bits of **hedeleg** are required specifically to be writable or zero, as enumerated in Table 128. Bit 0, corresponding to instruction address-misaligned exceptions, must be writable if $IALIGN=32$.



Requiring that certain bits of **hedeleg** be writable reduces some of the burden on a hypervisor to handle variations of implementation.

When $XLEN=32$, **hedelegh** is a 32-bit read/write register that aliases bits 63:32 of **hedeleg**. Register **hedelegh** does not exist when $XLEN=64$.

An interrupt that has been delegated to HS-mode (using **mideleg**) is further delegated to VS-mode if the corresponding **hideleg** bit is set. Among bits 15:0 of **hideleg**, bits 10, 6, and 2 (corresponding to the standard VS-level interrupts) are writable, and bits 12, 9, 5, and 1 (corresponding to the standard S-level interrupts) are read-only zeros.

When a virtual supervisor external interrupt (code 10) is delegated to VS-mode, it is automatically translated by the machine into a supervisor external interrupt (code 9) for VS-mode, including the value written to **vscause** on an interrupt trap. Likewise, a virtual supervisor timer interrupt (6) is translated into a supervisor timer interrupt (5) for VS-mode, and a virtual supervisor software interrupt (2) is translated into a supervisor software interrupt (1) for VS-mode. Similar translations may or may not be done for platform interrupt causes (codes 16 and above).

Table 128. Bits of **hedeleg** that must be writable or must be read-only zero.

Bit	Attribute	Corresponding Exception
0	(See text)	Instruction address misaligned
1	Writable	Instruction access fault
2	Writable	Illegal instruction
3	Writable	Breakpoint
4	Writable	Load address misaligned
5	Writable	Load access fault
6	Writable	Store/AMO address misaligned
7	Writable	Store/AMO access fault
8	Writable	Environment call from U-mode or VU-mode
9	Read-only 0	Environment call from HS-mode
10	Read-only 0	Environment call from VS-mode
11	Read-only 0	Environment call from M-mode
12	Writable	Instruction page fault
13	Writable	Load page fault
15	Writable	Store/AMO page fault
16	Read-only 0	Double trap
18	Writable	Software check
19	Writable	Hardware error
20	Read-only 0	Instruction guest-page fault
21	Read-only 0	Load guest-page fault
22	Read-only 0	Virtual instruction
23	Read-only 0	Store/AMO guest-page fault

5.2.3. Hypervisor Interrupt (**hvip**, **hip**, and **hie**) Registers

Register **hvip** is an HSXLEN-bit read/write register that a hypervisor can write to indicate virtual interrupts intended for VS-mode. Bits of **hvip** that are not writable are read-only zeros.



Figure 74. Hypervisor virtual-interrupt-pending register(**hvip**).

The standard portion (bits 15:0) of **hvip** is formatted as shown in Figure 75. Bits VSEIP, VSTIP, and VSSIP of **hvip** are writable. Setting VSEIP=1 in **hvip** asserts a VS-level external interrupt; setting VSTIP asserts a VS-level timer interrupt; and setting VSSIP asserts a VS-level software interrupt.

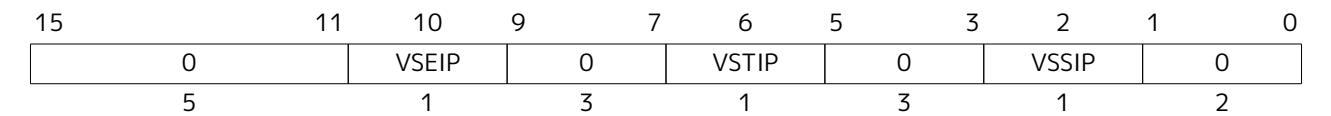
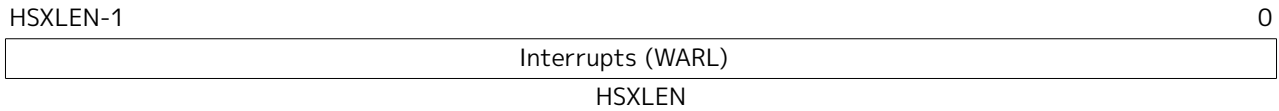
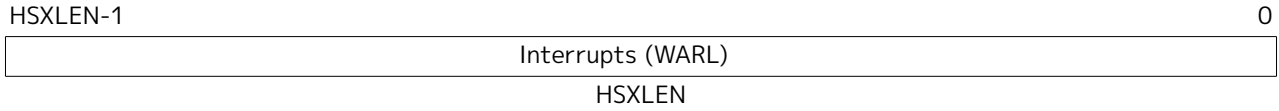


Figure 75. Standard portion (bits 15:0) of **hvip**.

Registers **hip** and **hie** are HSXLEN-bit read/write registers that supplement HS-level’s **sip** and **sie** respectively. The **hip** register indicates pending VS-level and hypervisor-specific interrupts, while **hie** contains enable bits for the same interrupts.

Figure 76. Hypervisor interrupt-pending register (**hip**).Figure 77. Hypervisor interrupt-enable register (**hie**).

For each writable bit in **sie**, the corresponding bit shall be read-only zero in both **hip** and **hie**. Hence, the nonzero bits in **sie** and **hie** are always mutually exclusive, and likewise for **sip** and **hip**.



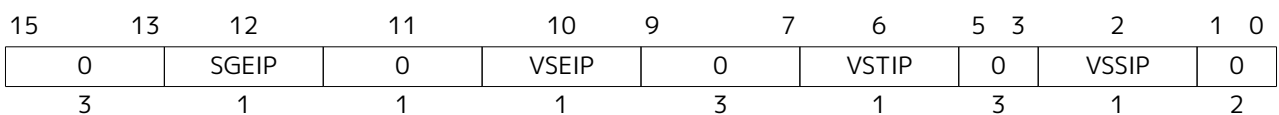
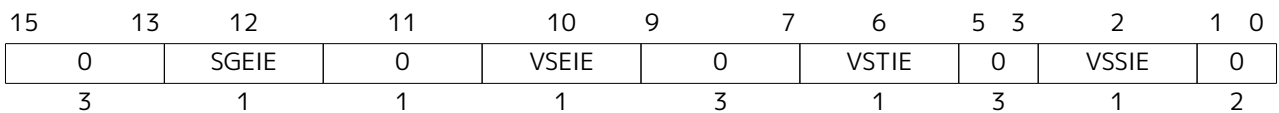
The active bits of **hip** and **hie** cannot be placed in HS-level's **sip** and **sie** because doing so would make it impossible for software to emulate the hypervisor extension on platforms that do not implement it in hardware.

An interrupt *i* will trap to HS-mode whenever all of the following are true: (a) either the current operating mode is HS-mode and the SIE bit in the **sstatus** register is set, or the current operating mode has less privilege than HS-mode; (b) bit *i* is set in both **sip** and **sie**, or in both **hip** and **hie**; and (c) bit *i* is not set in **hideleg**.

If bit *i* of **sie** is read-only zero, the same bit in register **hip** may be writable or may be read-only. When bit *i* in **hip** is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending in **hip** but bit *i* in **hip** is read-only, then either the interrupt can be cleared by clearing bit *i* of **hvip**, or the implementation must provide some other mechanism for clearing the pending interrupt (which may involve a call to the execution environment).

A bit in **hie** shall be writable if the corresponding interrupt can ever become pending in **hip**. Bits of **hie** that are not writable shall be read-only zero.

The standard portions (bits 15:0) of registers **hip** and **hie** are formatted as shown in Figure 78 and Figure 79 respectively.

Figure 78. Standard portion (bits 15:0) of **hip**.Figure 79. Standard portion (bits 15:0) of **hie**.

Bits **hip**.SGEIP and **hie**.SGEIE are the interrupt-pending and interrupt-enable bits for guest external interrupts at supervisor level (HS-level). SGEIP is read-only in **hip**, and is 1 if and only if the bitwise logical-AND of CSRs **hgeip** and **hgeie** is nonzero in any bit. (See Section 5.2.4.)

Bits **hip**.VSEIP and **hie**.VSEIE are the interrupt-pending and interrupt-enable bits for VS-level external interrupts. VSEIP is read-only in **hip**, and is the logical-OR of these interrupt sources:

- bit VSEIP of **hvip**;
- the bit of **hgeip** selected by **hstatus**.VGEIN; and

- any other platform-specific external interrupt signal directed to VS-level.

Bits **hip.VSTIP** and **hie.VSTIE** are the interrupt-pending and interrupt-enable bits for VS-level timer interrupts. VSTIP is read-only in **hip**, and is the logical-OR of **hvip.VSTIP** and, when the Sstc extension is implemented, the timer interrupt signal resulting from **vstimecmp**. The **hip.VSTIP** bit, in response to timer interrupts generated by **vstimecmp**, is set by writing **vstimecmp** with a value that is less than or equal to the sum of **time** and **htimedelta**, truncated to 64 bits; it is cleared by writing **vstimecmp** with a greater value. The **hip.VSTIP** bit remains defined while V=0 as well as V=1.

Bits **hip.VSSIP** and **hie.VSSIE** are the interrupt-pending and interrupt-enable bits for VS-level software interrupts. VSSIP in **hip** is an alias (writable) of the same bit in **hvip**.

Multiple simultaneous interrupts destined for HS-mode are handled in the following decreasing priority order: SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI.

5.2.4. Hypervisor Guest External Interrupt Registers (**hgeip** and **hgeie**)

The **hgeip** register is an HSXLEN-bit read-only register, formatted as shown in Figure 80, that indicates pending guest external interrupts for this hart. The **hgeie** register is an HSXLEN-bit read/write register, formatted as shown in Figure 81, that contains enable bits for the guest external interrupts at this hart. Guest external interrupt number *i* corresponds with bit *i* in both **hgeip** and **hgeie**.

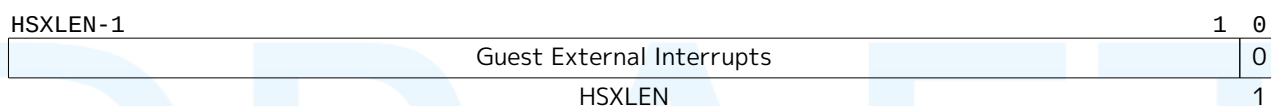


Figure 80. Hypervisor guest external interrupt-pending register (**hgeip**).



Figure 81. Hypervisor guest external interrupt-enable register (**hgeie**).

Guest external interrupts represent interrupts directed to individual virtual machines at VS-level. If a RISC-V platform supports placing a physical device under the direct control of a guest OS with minimal hypervisor intervention (known as *pass-through* or *direct assignment* between a virtual machine and the physical device), then, in such circumstance, interrupts from the device are intended for a specific virtual machine. Each bit of **hgeip** summarizes *all* pending interrupts directed to one virtual hart, as collected and reported by an interrupt controller. To distinguish specific pending interrupts from multiple devices, software must query the interrupt controller.



Support for guest external interrupts requires an interrupt controller that can collect virtual-machine-directed interrupts separately from other interrupts.

The number of bits implemented in **hgeip** and **hgeie** for guest external interrupts is UNSPECIFIED and may be zero. This number is known as *GEILEN*. The least-significant bits are implemented first, apart from bit 0. Hence, if *GEILEN* is nonzero, bits *GEILEN*:1 shall be writable in **hgeie**, and all other bit positions shall be read-only zeros in both **hgeip** and **hgeie**.



*The set of guest external interrupts received and handled at one physical hart may differ from those received at other harts. Guest external interrupt number *i* at one physical hart is typically expected not to be the same as guest external interrupt *i* at any other hart. For any one physical hart, the maximum number of virtual harts that may directly receive guest external interrupts is limited by *GEILEN*. The maximum this number can be for any implementation is 31 for RV32 and 63 for RV64, per physical hart.*

A hypervisor is always free to emulate devices for any number of virtual harts without being limited by GEILEN. Only direct pass-through (direct assignment) of interrupts is affected by the GEILEN limit, and the limit is on the number of virtual harts receiving such interrupts, not the number of distinct interrupts received. The number of distinct interrupts a single virtual hart may receive is determined by the interrupt controller.

Register **hgeie** selects the subset of guest external interrupts that cause a supervisor-level (HS-level) guest external interrupt. The enable bits in **hgeie** do not affect the VS-level external interrupt signal selected from **hgeip** by **hstatus.VGEIN**.

5.2.5. Hypervisor Environment Configuration Register (**henvcfg**)

The **henvcfg** CSR is a 64-bit read/write register, formatted as shown in [Figure 82](#), that controls certain characteristics of the execution environment when virtualization mode V=1.

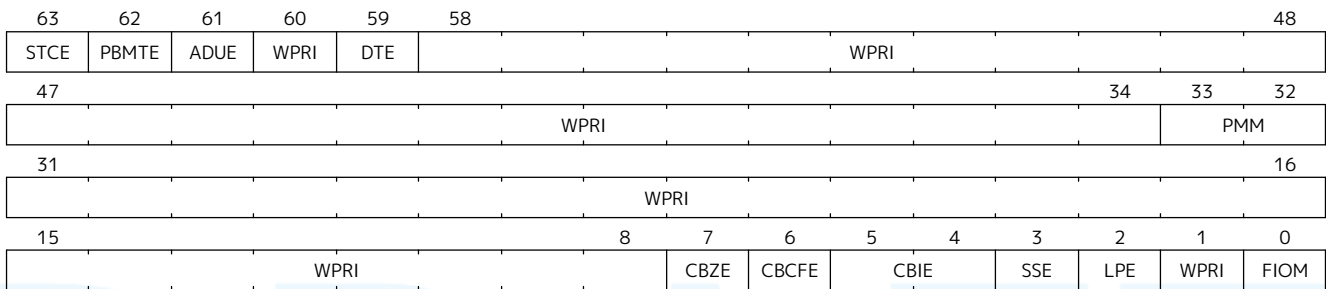


Figure 82. Hypervisor environment configuration register (**henvcfg**).

If bit FIOM (Fence of I/O implies Memory) is set to one in **henvcfg**, FENCE instructions executed when V=1 are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. [Table 129](#) details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO when FIOM=1 and V=1.

Similarly, when FIOM=1 and V=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

Table 129. Modified interpretation of FENCE predecessor and successor sets when FIOM=1 and virtualization mode V=1.

Instruction bit	Meaning when set
PI	Predecessor device input and memory reads (PR implied)
PO	Predecessor device output and memory writes (PW implied)
SI	Successor device input and memory reads (SR implied)
SO	Successor device output and memory writes (SW implied)

The PBMTE bit controls whether the Svpbmt extension is available for use in VS-stage address translation. When PBMTE=1, Svpbmt is available for VS-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented for VS-stage address translation. If Svpbmt is not implemented, PBMTE is read-only zero.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for VS-stage address translation. When ADUE=1, hardware updating of PTE A/D bits is enabled during VS-stage address translation, and the implementation behaves as though the Svade extension were not implemented for VS-mode address translation. When ADUE=0, the implementation behaves as though Svade were implemented for VS-stage address translation. If Svadu is not implemented, ADUE is read-only zero.

The Sstc extension adds the **STCE** (STimecmp Enable) bit to **henvcfg** CSR. When the Sstc extension is not implemented, **STCE** is read-only zero. The **STCE** bit enables **vstimecmp** for VS-mode when set to one. When **STCE** bit is **henvcfg** is zero, an attempt to access **stimecmp** (really **vstimecmp**) when V=1 raises a virtual-instruction exception, and **VSTIP** in **hip** reverts to its defined behavior as if this extension is not implemented.

The Zicboz extension adds the **CBZE** (Cache Block Zero instruction enable) field to **henvcfg**. The **CBZE** field applies to execution of the cache block zero instruction (**CB0.ZERO**) in privilege modes VS and VU, and only when the instruction is HS-qualified. If the instruction is not HS-qualified, it raises an illegal-instruction exception. If the instruction is HS-qualified and the **CBZE** field is set to 1, the instruction is enabled for execution; otherwise, if the **CBZE** field is set to 0, it raises a virtual-instruction exception. When the Zicboz extension is not implemented, **CBZE** is read-only zero.

The Zicbom extension adds the **CBCFE** (Cache Block Clean and Flush instruction Enable) field to **henvcfg**. When V=1, if the **CB0.CLEAN** and **CB0.FLUSH** instructions are not HS-qualified, they raise an illegal-instruction exception. If the instructions are HS-qualified and the **CBCFE** field is set to 1, the instructions are enabled for execution; otherwise, if the **CBCFE** field is set to 0, they raise a virtual-instruction exception. When the Zicbom extension is not implemented, **CBCFE** is read-only zero.

The Zicbom extension adds the **CBIE** (Cache Block Invalidate instruction Enable) WARL field to **henvcfg**. The **CBIE** field controls execution of the cache block invalidate instruction (**CB0.INVALID**) in privilege modes VS and VU. The encoding **10b** is reserved. When the Zicbom extension is not implemented, **CBIE** is read-only zero.

When V=1, if the **CB0.INVALID** instruction is not HS-qualified, it raises an illegal-instruction exception. If the instruction is HS-qualified and the **CBIE** field is set to **01b** or **11b**, the instruction is enabled for execution; otherwise, it raises a virtual-instruction exception.

If **CB0.INVALID** is enabled in HS-mode to perform a flush operation, then when the instruction is enabled in VS- or VU-mode it performs a flush operation, even if **CBIE** is set to **11b**. Otherwise, when the instruction is enabled for execution, its behavior depends on the **CBIE** encoding, as follows:

- **01b** — The instruction is executed and performs a flush operation, even if configured by VS-mode to perform an invalidate operation.
- **11b** — The instruction is executed and performs an invalidate operation, unless configured by VS-mode to perform a flush operation.

If the Ssnpm extension is implemented, the **PMM** field enables or disables pointer masking (see [Section 6.10](#)) for VS-mode, according to the values in [Table 130](#). When the Ssnpm extension is not implemented, the **PMM** field is read-only zero. The **PMM** field is read-only zero for RV32.

Table 130. Legal values of **PMM** WARL field

Value	Description
00	Pointer masking is disabled (PMLen = 0)
01	Reserved
10	Pointer masking is enabled with PMLen = XLen - 57 (PMLen = 7 on RV64)
11	Pointer masking is enabled with PMLen = XLen - 48 (PMLen = 16 on RV64)

The Zicfilp extension adds the **LPE** field in **henvcfg**. When the **LPE** field is set to 1, the Zicfilp extension is enabled in VS-mode. When the **LPE** field is 0, the Zicfilp extension is not enabled in VS-mode and the following rules apply to VS-mode:

- The hart does not update the **ELP** state; it remains as **NO_LP_EXPECTED**.

- The **LPAD** instruction operates as a no-op.

The Zicfiss extension adds the **SSE** field in **henvcfg**. If the **SSE** field is set to 1, the Zicfiss extension is activated in VS-mode. When the **SSE** field is 0, the Zicfiss extension remains inactive in VS-mode, and the following rules apply when **V=1**:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS-stage page tables becomes reserved.
- The `senvcfg.SSE` field will read as zero and is read-only.
- When `menvcfg.SSE` is one, `SSAMOSWAP.W/D` raises a virtual-instruction exception.

The Ssdbltrp extension adds the double-trap-enable (DTE) field in `henvcfg`. When `henvcfg.DTE` is zero, the implementation behaves as though Ssdbltrp is not implemented for VS-mode and the `vsstatus.SDT` bit is read-only zero.

When XLEN=32, **henvcfgh** is a 32-bit read/write register that aliases bits 63:32 of **henvcfg**. Register **henvcfgh** does not exist when XLEN=64.

5.2.6. Hypervisor Counter-Enable (hcounteren) Register

The counter-enable register **hcounteren** is a 32-bit register that controls the availability of the hardware performance monitoring counters to the guest virtual machine.

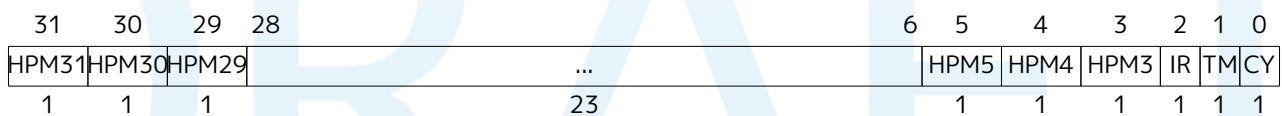


Figure 83. Hypervisor counter-enable register (hcounteren).

When the CY, TM, IR, or HPM n bit in the **hcounter n** register is clear, attempts to read the **cycle**, **time**, **instret**, or **hpmcounter n** register while V=1 will cause a virtual-instruction exception if the same bit in **mcounter n** is 1. When one of these bits is set, access to the corresponding register is permitted when V=1, unless prevented for some other reason. In VU-mode, a counter is not readable unless the applicable bits are set in both **hcounter n** and **scounter n** .

In addition, when the TM bit in the `hcounteren` register is clear, attempts to access the `vstimecmp` register (via `stimecmp`) while executing in VS-mode will cause a virtual-instruction exception if the same bit in `mcounteren` is set. When this bit and the same bit in `mcounteren` are both set, access to the `vstimecmp` register (if implemented) is permitted in VS-mode.

hcounteren must be implemented. However, any of the bits may be read-only zero, indicating reads to the corresponding counter will cause an exception when $V=1$. Hence, they are effectively **WARL** fields.

5.2.7. Hypervisor Time Delta (htimedelta) Register

The **htimedelta** CSR is a 64-bit read/write register that contains the delta between the value of the **time** CSR and the value returned in VS-mode or VU-mode. That is, reading the **time** CSR in VS or VU mode returns the sum of the contents of **htimedelta** and the actual value of **time**.



Because overflow is ignored when summing `htimedelta` and `time`, large values of `htimedelta` may be used to represent negative time offsets.

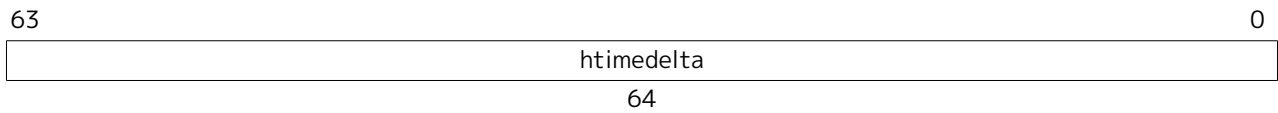


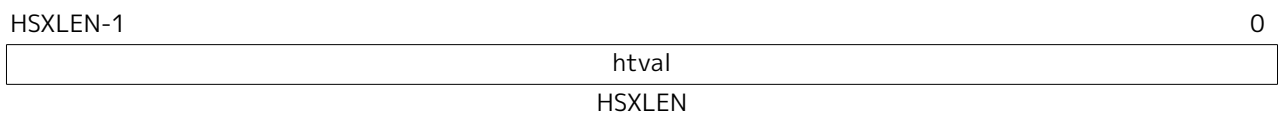
Figure 84. Hypervisor time delta register.

When $XLEN=32$, **htimedeltah** is a 32-bit read/write register that aliases bits 63:32 of **htimedelta**. Register **htimedeltah** does not exist when $XLEN=64$.

If the **time** CSR is implemented, **htimedelta** (and **htimedeltah** for $XLEN=32$) must be implemented.

5.2.8. Hypervisor Trap Value (**htval**) Register

The **htval** register is an $HSXLEN$ -bit read/write register formatted as shown in Figure 85. When a trap is taken into HS-mode, **htval** is written with additional exception-specific information, alongside **stval**, to assist software in handling the trap.

Figure 85. Hypervisor trap value register (**htval**).

When a guest-page-fault trap is taken into HS-mode, **htval** is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, **htval** is set to zero, but a future standard or extension may redefine **htval**'s setting for other traps.

A guest-page fault may arise due to an implicit memory access during first-stage (VS-stage) address translation, in which case a guest physical address written to **htval** is that of the implicit memory access that faulted—for example, the address of a VS-level page table entry that could not be read. (The guest physical address corresponding to the original virtual address is unknown when VS-stage translation fails to complete.) Additional information is provided in CSR **htinst** to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in **htval** corresponds to the faulting portion of the access as indicated by the virtual address in **stval**. For instruction guest-page faults on systems with variable-length instructions, a nonzero **htval** corresponds to the faulting portion of the instruction as indicated by the virtual address in **stval**.



*A guest physical address written to **htval** is shifted right by 2 bits to accommodate addresses wider than the current $XLEN$. For RV32, the hypervisor extension permits guest physical addresses as wide as 34 bits, and **htval** reports bits 33:2 of the address. This shift-by-2 encoding of guest physical addresses matches the encoding of physical addresses in PMP address registers (Section 3.7) and in page table entries (Section 4.3, Section 4.4, Section 4.5, and Section 4.6).*

*If the least-significant two bits of a faulting guest physical address are needed, these bits are ordinarily the same as the least-significant two bits of the faulting virtual address in **stval**. For faults due to implicit memory accesses for VS-stage address translation, the least-significant two bits are instead zeros. These cases can be distinguished using the value provided in register **htinst**.*

htval is a **WARL** register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.



*Unless it has reason to assume otherwise (such as a platform standard), software that writes a value to **htval** should read back from **htval** to confirm the stored value.*

5.2.9. Hypervisor Trap Instruction (**htinst**) Register

The **htinst** register is an HSXLEN-bit read/write register formatted as shown in Figure 86. When a trap is taken into HS-mode, **htinst** is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to **htinst** on a trap are documented in Section 5.6.3.

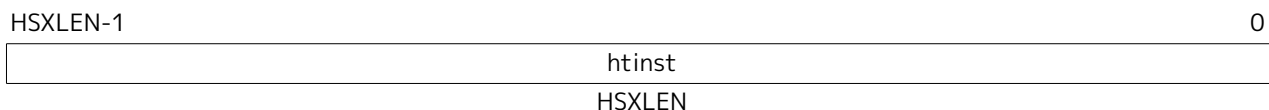


Figure 86. Hypervisor trap instruction (**htinst**) register.

htinst is a WARL register that need only be able to hold the values that the implementation may automatically write to it on a trap.

5.2.10. Hypervisor Guest Address Translation and Protection (**hgap**) Register

The **hgap** register is an HSXLEN-bit read/write register, formatted as shown in Figure 87 for HSXLEN=32 and Figure 88 for HSXLEN=64, which controls G-stage address translation and protection, the second stage of two-stage translation for guest virtual addresses (see Section 5.5). Similar to CSR **satp**, this register holds the physical page number (PPN) of the guest-physical root page table; a virtual machine identifier (VMID), which facilitates address-translation fences on a per-virtual-machine basis; and the MODE field, which selects the address-translation scheme for guest physical addresses. When **mstatus.TVM**=1, attempts to read or write **hgap** while executing in HS-mode will raise an illegal-instruction exception.

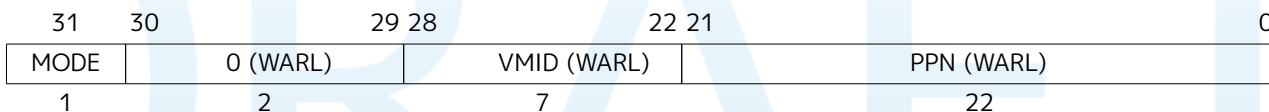


Figure 87. Hypervisor guest address translation and protection register **hgap** when HSXLEN=32.

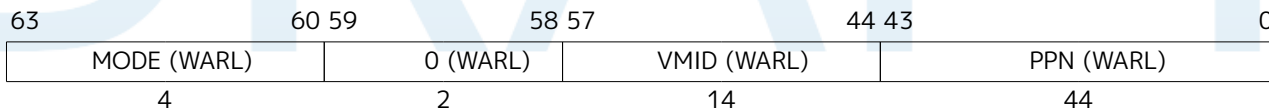


Figure 88. Hypervisor guest address translation and protection register **hgap** when HSXLEN=64 for MODE values Bare, Sv39x4, Sv48x4, and Sv57x4.

Table 131 shows the encodings of the MODE field when HSXLEN=32 and HSXLEN=64. When MODE=Bare, guest physical addresses are equal to supervisor physical addresses, and there is no further memory protection for a guest virtual machine beyond the physical memory protection scheme described in Section 3.7. In this case, software must write zero to the remaining fields in **hgap**. Attempting to select MODE=Bare with a nonzero pattern in the remaining fields has an UNSPECIFIED effect on the value that the remaining fields assume and an UNSPECIFIED effect on G-stage address translation and protection behavior.

When HSXLEN=32, the only other valid setting for MODE is Sv32x4, which is a modification of the usual Sv32 paged virtual-memory scheme, extended to support 34-bit guest physical addresses. When HSXLEN=64, modes Sv39x4, Sv48x4, and Sv57x4 are defined as modifications of the Sv39, Sv48, and Sv57 paged virtual-memory schemes. All of these paged virtual-memory schemes are described in Section 5.5.1.

The remaining MODE settings when HSXLEN=64 are reserved for future use and may define different interpretations of the other fields in **hgap**.

Table 131. Encoding of **hgap** MODE field.

HSXLEN=32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32x4	Page-based 34-bit virtual addressing (2-bit extension of Sv32).
HSXLEN=64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	—	<i>Reserved</i>
8	Sv39x4	Page-based 41-bit virtual addressing (2-bit extension of Sv39).
9	Sv48x4	Page-based 50-bit virtual addressing (2-bit extension of Sv48).
10	Sv57x4	Page-based 59-bit virtual addressing (2-bit extension of Sv57).
11-15	—	<i>Reserved</i>

Implementations are not required to support all defined MODE settings when HSXLEN=64.

A write to **hgap** with an unsupported MODE value is not ignored as it is for **satp**. Instead, the fields of **hgap** are **WARL** in the normal way, when so indicated.

As explained in [Section 5.5.1](#), for the paged virtual-memory schemes (Sv32x4, Sv39x4, Sv48x4, and Sv57x4), the root page table is 16 KiB and must be aligned to a 16-KiB boundary. In these modes, the lowest two bits of the physical page number (PPN) in **hgap** always read as zeros. An implementation that supports only the defined paged virtual-memory schemes and/or Bare may make PPN[1:0] read-only zero.

The number of VMID bits is UNSPECIFIED and may be zero. The number of implemented VMID bits, termed *VMIDLEN*, may be determined by writing one to every bit position in the VMID field, then reading back the value in **hgap** to see which bit positions in the VMID field hold a one. The least-significant bits of VMID are implemented first: that is, if *VMIDLEN* > 0, VMID[VMIDLEN-1:0] is writable. The maximal value of VMIDLEN, termed VMIDMAX, is 7 for Sv32x4 or 14 for Sv39x4, Sv48x4, and Sv57x4.

The **hgap** register is considered *active* for the purposes of the address-translation algorithm *unless* the effective privilege mode is U and **hstatus.HU**=0.



This definition simplifies the implementation of speculative execution of HLV, HLVX, and HSV instructions.

Note that writing **hgap** does not imply any ordering constraints between page-table updates and subsequent G-stage address translations. If the new virtual machine's guest physical page tables have been modified, or if a VMID is reused, it may be necessary to execute an HFENCE.GVMA instruction (see [Section 5.3.2](#)) before or after writing **hgap**.

5.2.11. Virtual Supervisor Status (**vsstatus**) Register

The **vsstatus** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **sstatus**, formatted as shown in [Figure 89](#) when VSXLEN=32 and [Figure 90](#) when VSXLEN=64. When V=1, **vsstatus** substitutes for the usual **sstatus**, so instructions that normally read or modify **sstatus** actually access **vsstatus** instead.

31	30					25	24	23	22		20	19	18	17	16
SD						WPRI	SDT	SPELP		WPRI		MXR	SUM	WPRI	XS[1:0]
15	14	13	12	11	10	9	8	7	6	5	4		2	1	0
XS[1:0]		FS[1:0]		WPRI		VS[1:0]	SPP	WPRI	UBE	SPIE		WPRI		SIE	WPRI

Figure 89. Virtual supervisor status (**vsstatus**) register when VSXLEN=32.

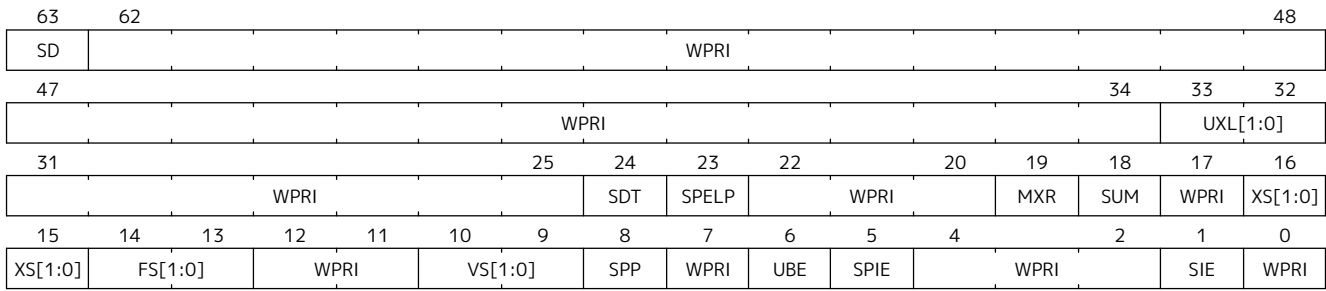


Figure 90. Virtual supervisor status (**vsstatus**) register when VSXLEN=64.

The UXL field controls the effective XLEN for VU-mode, which may differ from the XLEN for VS-mode (VSXLEN). When VSXLEN=32, the UXL field does not exist, and VU-mode XLEN=32. When VSXLEN=64, UXL is a **WARL** field that is encoded the same as the MXL field of **misal**, shown in Table 107. In particular, an implementation may make UXL be a read-only copy of field VSXL of **hstatus**, forcing VU-mode XLEN=VSXLEN.

If VSXLEN is changed from 32 to a wider width, and if field UXL is not restricted to a single value, it gets the value corresponding to the widest supported width not wider than the new VSXLEN.

When V=1, both **vsstatus.FS** and the HS-level **sstatus.FS** are in effect. Attempts to execute a floating-point instruction when either field is 0 (Off) raise an illegal-instruction exception. Modifying the floating-point state when V=1 causes both fields to be set to 3 (Dirty).



For a hypervisor to benefit from the extension context status, it must have its own copy in the HS-level **sstatus**, maintained independently of a guest OS running in VS-mode. While a version of the extension context status obviously must exist in **vsstatus** for VS-mode, a hypervisor cannot rely on this version being maintained correctly, given that VS-level software can change **vsstatus.FS** arbitrarily. If the HS-level **sstatus.FS** were not independently active and maintained by the hardware in parallel with **vsstatus.FS** while V=1, hypervisors would always be forced to conservatively swap all floating-point state when context-switching between virtual machines.

Similarly, when V=1, both **vsstatus.VS** and the HS-level **sstatus.VS** are in effect. Attempts to execute a vector instruction when either field is 0 (Off) raise an illegal-instruction exception. Modifying the vector state when V=1 causes both fields to be set to 3 (Dirty).

Read-only fields SD and XS summarize the extension context status as it is visible to VS-mode only. For example, the value of the HS-level **sstatus.FS** does not affect **vsstatus.SD**.

An implementation may make field UBE be a read-only copy of **hstatus.VSBE**.

When V=0, **vsstatus** does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVS, or HSV) or the MPRV feature in the **mstatus** register is used to execute a load or store as though V=1.

The Zicfilp extension adds the **SPELP** field that holds the previous **ELP**, and is updated as specified in Section 6.9.1.2. The **SPELP** field is encoded as follows:

- 0 - **NO_LP_EXPECTED** - no landing pad instruction expected.
- 1 - **LP_EXPECTED** - a landing pad instruction is expected.

The Ssdbltrap adds an S-mode-disable-trap (**SDT**) field extension to address double trap (See Section 4.1.1.5) in VS-mode.

5.2.12. Virtual Supervisor Interrupt (**vsip** and **vsie**) Registers

The **vsip** and **vsie** registers are VSXLEN-bit read/write registers that are VS-mode's versions of supervisor CSRs **sip** and **sie**, formatted as shown in Figure 91 and Figure 92 respectively. When V=1, **vsip** and **vsie** substitute for the usual **sip** and **sie**, so instructions that normally read or modify **sip**/**sie** actually access **vsip**/**vsie** instead. However, interrupts directed to HS-level continue to be indicated in the HS-level **sip** register, not in **vsip**, when V=1.

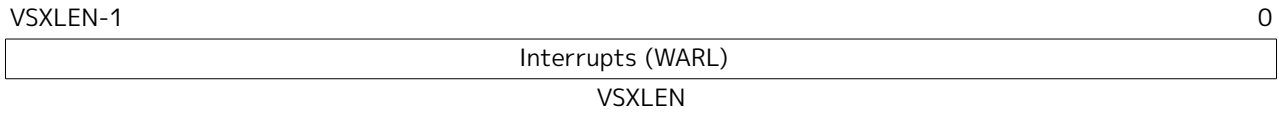


Figure 91. Virtual supervisor interrupt-pending register (**vsip**).

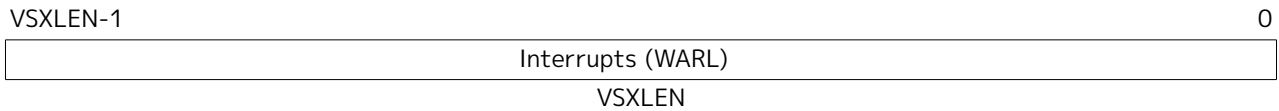


Figure 92. Virtual supervisor interrupt-enable register (**vsie**).

The standard portions (bits 15:0) of registers **vsip** and **vsie** are formatted as shown in Figure 93 and Figure 94 respectively.

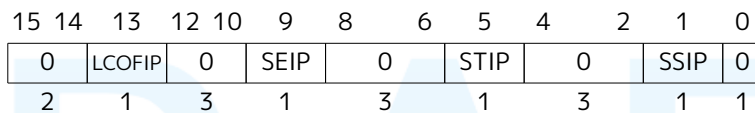


Figure 93. Standard portion (bits 15:0) of **vsip**.

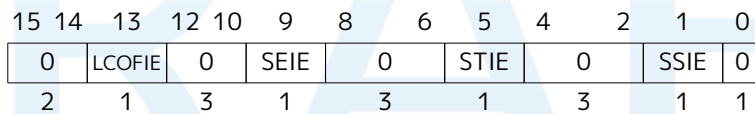


Figure 94. Standard portion (bits 15:0) of **vsie**.

Extension Shlcofideleg supports delegating LCOFI interrupts to VS-mode. If the Shlcofideleg extension is implemented, **hideleg** bit 13 is writable; otherwise, it is read-only zero. When bit 13 of **hideleg** is zero, **vsip**.LCOFIP and **vsie**.LCOFIE are read-only zeros. Else, **vsip**.LCOFIP and **vsie**.LCOFIE are aliases of **sip**.LCOFIP and **sie**.LCOFIE.

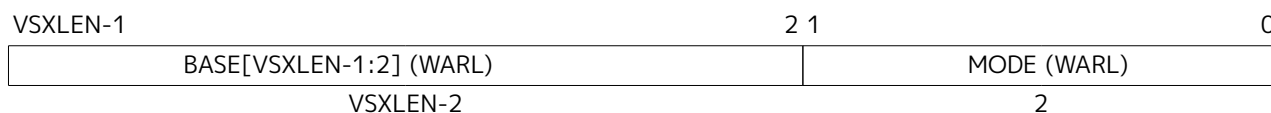
When bit 10 of **hideleg** is zero, **vsip**.SEIP and **vsie**.SEIE are read-only zeros. Else, **vsip**.SEIP and **vsie**.SEIE are aliases of **hip**.VSEIP and **hie**.VSEIE.

When bit 6 of **hideleg** is zero, **vsip**.STIP and **vsie**.STIE are read-only zeros. Else, **vsip**.STIP and **vsie**.STIE are aliases of **hip**.VSTIP and **hie**.VSTIE.

When bit 2 of **hideleg** is zero, **vsip**.SSIP and **vsie**.SSIE are read-only zeros. Else, **vsip**.SSIP and **vsie**.SSIE are aliases of **hip**.VSSIP and **hie**.VSSIE.

5.2.13. Virtual Supervisor Trap Vector Base Address (**vstvec**) Register

The **vstvec** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **stvec**, formatted as shown in Figure 95. When V=1, **vstvec** substitutes for the usual **stvec**, so instructions that normally read or modify **stvec** actually access **vstvec** instead. When V=0, **vstvec** does not directly affect the behavior of the machine.

Figure 95. Virtual supervisor trap vector base address register **vstvec**.

5.2.14. Virtual Supervisor Scratch (**vsscratch**) Register

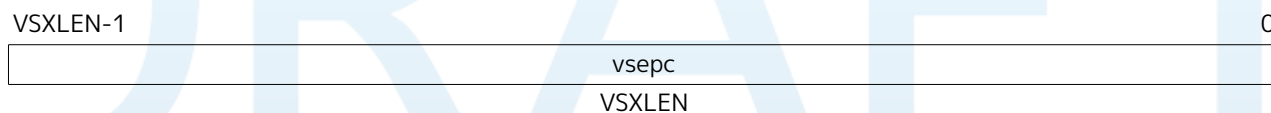
The **vsscratch** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **sscratch**, formatted as shown in Figure 96. When $V=1$, **vsscratch** substitutes for the usual **sscratch**, so instructions that normally read or modify **sscratch** actually access **vsscratch** instead. The contents of **vsscratch** never directly affect the behavior of the machine.

Figure 96. Virtual supervisor scratch register **vsscratch**.

5.2.15. Virtual Supervisor Exception Program Counter (**vsepc**) Register

The **vsepc** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **sepc**, formatted as shown in Figure 97. When $V=1$, **vsepc** substitutes for the usual **sepc**, so instructions that normally read or modify **sepc** actually access **vsepc** instead. When $V=0$, **vsepc** does not directly affect the behavior of the machine.

vsepc is a WARL register that must be able to hold the same set of values that **sepc** can hold.

Figure 97. Virtual supervisor exception program counter (**vsepc**).

5.2.16. Virtual Supervisor Cause (**vscause**) Register

The **vscause** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **scause**, formatted as shown in Figure 98. When $V=1$, **vscause** substitutes for the usual **scause**, so instructions that normally read or modify **scause** actually access **vscause** instead. When $V=0$, **vscause** does not directly affect the behavior of the machine.

vscause is a WLRL register that must be able to hold the same set of values that **scause** can hold.

Figure 98. Virtual supervisor cause register (**vscause**).

5.2.17. Virtual Supervisor Trap Value (**vstval**) Register

The **vstval** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **stval**, formatted as shown in Figure 99. When $V=1$, **vstval** substitutes for the usual **stval**, so instructions that normally read or modify **stval** actually access **vstval** instead. When $V=0$, **vstval** does not directly

affect the behavior of the machine.

vstval is a **WARL** register that must be able to hold the same set of values that **stval** can hold.

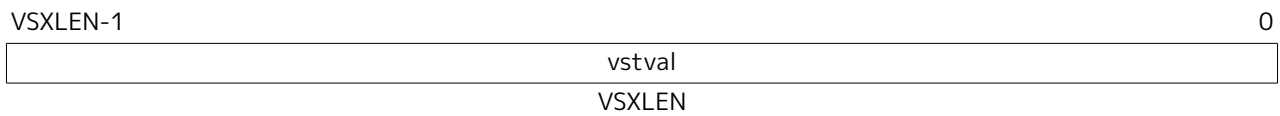


Figure 99. Virtual supervisor trap value register (**vstval**).

5.2.18. Virtual Supervisor Address Translation and Protection (**vsatp**) Register

The **vsatp** register is a VSXLEN-bit read/write register that is VS-mode's version of supervisor register **satp**, formatted as shown in Figure 100 for VSXLEN=32 and Figure 101 for VSXLEN=64. When V=1, **vsatp** substitutes for the usual **satp**, so instructions that normally read or modify **satp** actually access **vsatp** instead. **vsatp** controls VS-stage address translation, the first stage of two-stage translation for guest virtual addresses (see Section 5.5).

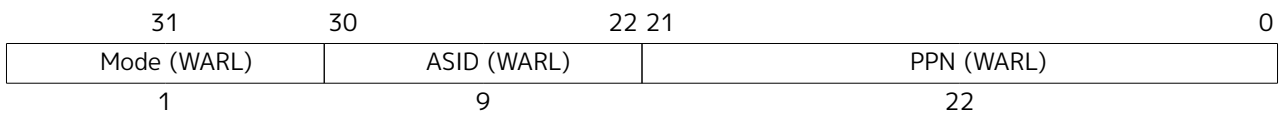


Figure 100. Virtual supervisor address translation and protection **vsatp** register when VSXLEN=32.

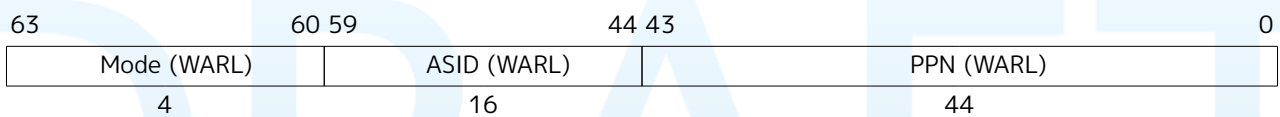


Figure 101. Virtual supervisor address translation and protection **vsatp** register when VSXLEN=64.

The **vsatp** register is considered *active* for the purposes of the address-translation algorithm *unless* the effective privilege mode is U and **hstatus**.HU=0. However, even when **vsatp** is active, VS-stage page-table entries' A bits must not be set as a result of speculative execution, unless the effective privilege mode is VS or VU.



In particular, virtual-machine load/store (HLV, HLVX, or HSV) instructions that are mispredicted must not cause VS-stage A bits to be set.

When V=0, a write to **vsatp** with an unsupported MODE value is either ignored as it is for **satp**, or the fields of **vsatp** are treated as **WARL** in the normal way. However, when V=1, a write to **satp** with an unsupported MODE value is ignored and no write to **vsatp** is effected.

When V=0, **vsatp** does not directly affect the behavior of the machine, unless a virtual-machine load/store (HLV, HLVX, or HSV) or the MPRV feature in the **mstatus** register is used to execute a load or store as though V=1.

5.2.19. Virtual Supervisor Timer (**vstimecmp**) Register

The **vstimecmp** CSR is a 64-bit register and has 64-bit precision on all RV32 and RV64 systems. In RV32 only, accesses to the **vstimecmp** CSR access the low 32 bits, while accesses to the **vstimecmph** CSR access the high 32 bits of **vstimecmp**.

A virtual supervisor timer interrupt becomes pending, as reflected in the VSTIP bit in the **hiep** register, whenever $(\text{time} + \text{htimedelta})$, truncated to 64 bits, contains a value greater than or equal to **vstimecmp**, treating the values as unsigned integers. If the result of this comparison changes, it is guaranteed to be

reflected in VSTIP eventually, but not necessarily immediately. The interrupt remains posted until `vstimecmp` becomes greater than `(time + htimedelta)`, typically as a result of writing `vstimecmp`. The interrupt will be taken based on the standard interrupt enable and delegation rules while `V=1`.

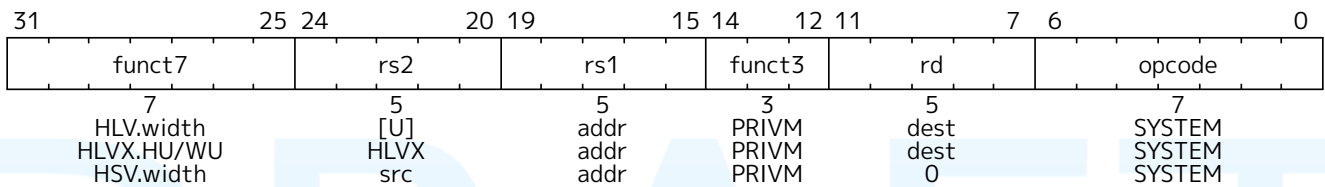


In systems in which a supervisor execution environment (SEE) implemented by an HS-mode hypervisor provides timer facilities via an SBI function call, this SBI call will continue to support requests to schedule a timer interrupt. The SEE will simply make use of `vstimecmp`, changing its value as appropriate. This ensures compatibility with existing guest VS-mode software that uses this SEE facility, while new VS-mode software takes advantage of `vstimecmp` directly.)

5.3. Hypervisor Instructions

The hypervisor extension adds virtual-machine load and store instructions and two privileged fence instructions.

5.3.1. Hypervisor Virtual-Machine Load and Store Instructions



The hypervisor virtual-machine load and store instructions are valid only in M-mode or HS-mode, or in U-mode when `hstatus.HU=1`. Each instruction performs an explicit memory access with an effective privilege mode of VS or VU. The effective privilege mode of the explicit memory access is VU when `hstatus.SPVP=0`, and VS when `hstatus.SPVP=1`. As usual for VS-mode and VU-mode, two-stage address translation is applied, and the HS-level `sstatus.SUM` is ignored. HS-level `sstatus.MXR` makes execute-only pages readable by explicit loads for both stages of address translation (VS-stage and G-stage), whereas `vsstatus.MXR` affects only the first translation stage (VS-stage).

For every RV32I or RV64I load instruction, LB, LBU, LH, LHU, LW, LWU, and LD, there is a corresponding virtual-machine load instruction: HLV.B, HLV.BU, HLV.H, HLV.HU, HLV.W, HLV.WU, and HLV.D. For every RV32I or RV64I store instruction, SB, SH, SW, and SD, there is a corresponding virtual-machine store instruction: HSV.B, HSV.H, HSV.W, and HSV.D. Instructions HLV.WU, HLV.D, and HSV.D are not valid for RV32, of course.

Instructions HLVX.HU and HLVX.WU are the same as HLV.HU and HLV.WU, except that *execute* permission takes the place of *read* permission during address translation. That is, the memory being read must be executable in both stages of address translation, but read permission is not required. For the supervisor physical address that results from address translation, the supervisor physical memory attributes must grant both *execute* and *read* permissions. (The *supervisor physical memory attributes* are the machine's physical memory attributes as modified by physical memory protection, [Section 3.7](#), for supervisor level.)



HLVX cannot override machine-level physical memory protection (PMP), so attempting to read memory that PMP designates as execute-only still results in an access-fault exception.

Although HLVX instructions' explicit memory accesses require execute permissions, they still raise the same exceptions as other load instructions, rather than raising fetch exceptions instead.

HLVX.WU is valid for RV32, even though LWU and HLV.WU are not. (For RV32, HLVX.WU can be

considered a variant of `HLV.W`, as sign extension is irrelevant for 32-bit values.)

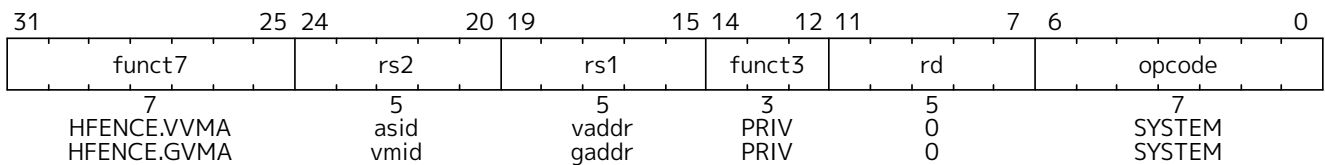
The memory accesses performed by the `HLVX.*` instructions are not subject to pointer masking (see [Section 6.10](#)).



`HLVX.*` instructions, designed for emulating implicit access to fetch instructions from guest memory, perform memory accesses that are exempt from pointer masking to facilitate this emulation. For the same reason, pointer masking does not apply when `MXR` is set.

Attempts to execute a virtual-machine load/store instruction (`HLV`, `HLVX`, or `HSV`) when `V=1` cause a virtual-instruction exception. Attempts to execute one of these same instructions from U-mode when `hstatus.HU=0` cause an illegal-instruction exception.

5.3.2. Hypervisor Memory-Management Fence Instructions



The hypervisor memory-management fence instructions, `HFENCE.VVMA` and `HFENCE.GVMA`, perform a function similar to `SFENCE.VMA` ([Section 4.2.1](#)), except applying to the VS-level memory-management data structures controlled by CSR `vsatp` (`HFENCE.VVMA`) or the guest-physical memory-management data structures controlled by CSR `hgatp` (`HFENCE.GVMA`). Instruction `SFENCE.VMA` applies only to the memory-management data structures controlled by the current `satp` (either the HS-level `satp` when `V=0` or `vsatp` when `V=1`).

`HFENCE.VVMA` is valid only in M-mode or HS-mode. Its effect is much the same as temporarily entering VS-mode and executing `SFENCE.VMA`. Executing an `HFENCE.VVMA` guarantees that any previous stores already visible to the current hart are ordered before all implicit reads by that hart done for VS-stage address translation for instructions that

- are subsequent to the `HFENCE.VVMA`, and
- execute when `hgatp.VMID` has the same setting as it did when `HFENCE.VVMA` executed.

Implicit reads need not be ordered when `hgatp.VMID` is different than at the time `HFENCE.VVMA` executed. If operand `rs1` $\neq$ `x0`, it specifies a single guest virtual address, and if operand `rs2` $\neq$ `x0`, it specifies a single guest address-space identifier (ASID).



An `HFENCE.VVMA` instruction applies only to a single virtual machine, identified by the setting of `hgatp.VMID` when `HFENCE.VVMA` executes.

When `rs2` $\neq$ `x0`, bits `XLEN-1:ASIDMAX` of the value held in `rs2` are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if `ASIDLEN` $<$ `ASIDMAX`, the implementation shall ignore bits `ASIDMAX-1:ASIDLEN` of the value held in `rs2`.



Simpler implementations of `HFENCE.VVMA` can ignore the guest virtual address in `rs1` and the guest ASID value in `rs2`, as well as `hgatp.VMID`, and always perform a global fence for the VS-level memory management of all virtual machines, or even a global fence for all memory-management data structures.

Neither `mstatus.TVM` nor `hstatus.VTVM` causes `HFENCE.VVMA` to trap.

HFENCE.GVMA is valid only in HS-mode when `mstatus.TVM=0`, or in M-mode (irrespective of `mstatus.TVM`). Executing an HFENCE.GVMA instruction guarantees that any previous stores already visible to the current hart are ordered before all implicit reads by that hart done for G-stage address translation for instructions that follow the HFENCE.GVMA. If operand `rs1≠x0`, it specifies a single guest physical address, shifted right by 2 bits, and if operand `rs2≠x0`, it specifies a single virtual machine identifier (VMID).

Conceptually, an implementation might contain two address-translation caches: one that maps guest virtual addresses to guest physical addresses, and another that maps guest physical addresses to supervisor physical addresses. HFENCE.GVMA need not flush the former cache, but it must flush entries from the latter cache that match the HFENCE.GVMA's address and VMID arguments.

More commonly, implementations contain address-translation caches that map guest virtual addresses directly to supervisor physical addresses, removing a level of indirection. For such implementations, any entry whose guest virtual address maps to a guest physical address that matches the HFENCE.GVMA's address and VMID arguments must be flushed. Selectively flushing entries in this fashion requires tagging them with the guest physical address, which is costly, and so a common technique is to flush all entries that match the HFENCE.GVMA's VMID argument, regardless of the address argument.

Like for a guest physical address written to `htval` on a trap, a guest physical address specified in `rs1` is shifted right by 2 bits to accommodate addresses wider than the current XLEN.

When `rs2≠x0`, bits XLEN-1:VMIDMAX of the value held in `rs2` are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if `VMIDLEN < VMIDMAX`, the implementation shall ignore bits VMIDMAX-1:VMIDLEN of the value held in `rs2`.

Simpler implementations of HFENCE.GVMA can ignore the guest physical address in `rs1` and the VMID value in `rs2` and always perform a global fence for the guest-physical memory management of all virtual machines, or even a global fence for all memory-management data structures.

If `hgap.MODE` is changed for a given VMID, an HFENCE.GVMA with `rs1=x0` (and `rs2` set to either `x0` or the VMID) must be executed to order subsequent guest translations with the MODE change—even if the old MODE or new MODE is Bare.

Attempts to execute HFENCE.VVMA or HFENCE.GVMA when `V=1` cause a virtual-instruction exception, while attempts to do the same in U-mode cause an illegal-instruction exception. Attempting to execute HFENCE.GVMA in HS-mode when `mstatus.TVM=1` also causes an illegal-instruction exception.

5.4. Machine-Level CSRs

The hypervisor extension augments or modifies machine CSRs `mstatus`, `mstatush`, `mideleg`, `mip`, and `mie`, and adds CSRs `mtval2` and `mtinst`.

5.4.1. Machine Status (`mstatus` and `mstatush`) Registers

The hypervisor extension adds two fields, MPV and GVA, to the machine-level `mstatus` or `mstatush` CSR, and modifies the behavior of several existing `mstatus` fields. Figure 102 shows the modified `mstatus` register when the hypervisor extension is implemented and MXLEN=64. When MXLEN=32, the hypervisor

extension adds MPV and GVA not to **mstatus** but to **mstatush**. Figure 103 shows the **mstatush** register when the hypervisor extension is implemented and MXLEN=32.

63	62		40	39	38	37	36	35	34	33	32
SD	WPRI			MPV	GVA	MBE	SBE	SXL[1:0]		UXL[1:0]	
1	23			1	1	1	1	2		2	
31		23	22	21	20	19	18	17	16	15	14 13
	WPRI		TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]	FS[1:0]	
9		1	1	1	1	1	1	1	2	2	
12	11	10	9	8	7	6	5	4	3	2	1 0
	MPP[1:0]	VS[1:0]	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI
2	2	1	1	1	1	1	1	1	1	1	1

Figure 102. Machine status (**mstatus**) register for RV64 when the hypervisor extension is implemented.

31	8	7	6	5	4	3	0
WPRI	MPV	GVA	MBE	SBE	WPRI		
24	1	1	1	1			4

Figure 103. Additional machine status (**mstatush**) register for RV32 when the hypervisor extension is implemented.
The format of **mstatus** is unchanged for RV32.

The MPV bit (Machine Previous Virtualization Mode) is written by the implementation whenever a trap is taken into M-mode. Just as the MPP field is set to the (nominal) privilege mode at the time of the trap, the MPV bit is set to the value of the virtualization mode V at the time of the trap. When an MRET instruction is executed, the virtualization mode V is set to MPV, unless MPP=3, in which case V remains 0.

Field GVA (Guest Virtual Address) is written by the implementation whenever a trap is taken into M-mode. For any trap (breakpoint, address misaligned, access fault, page fault, or guest-page fault) that writes a guest virtual address to **mtval**, GVA is set to 1. For any other trap into M-mode, GVA is set to 0.

The TSR and TVM fields of **mstatus** affect execution only in HS-mode, not in VS-mode. The TW field affects execution in all modes except M-mode.

Setting TVM=1 prevents HS-mode from accessing **hgap** or executing HFENCE.GVMA or HINVAL.GVMA, but has no effect on accesses to **vsatp** or instructions HFENCE.VVMA or HINVAL.VVMA.



*TVM exists in **mstatus** to allow machine-level software to modify the address translations managed by a supervisor-level OS, usually for the purpose of inserting another stage of address translation below that controlled by the OS. The instruction traps enabled by TVM=1 permit machine level to co-opt both **satp** and **hgap** and substitute shadow page tables that merge the OS's chosen page translations with M-level's lower-stage translations, all without the OS being aware. M-level software needs this ability not only to emulate the hypervisor extension if not already supported, but also to emulate any future RISC-V extensions that may modify or add address translation stages, perhaps, for example, to improve support for nested hypervisors, i.e., running hypervisors atop other hypervisors.*

*However, setting TVM=1 does not cause traps for accesses to **vsatp** or instructions HFENCE.VVMA or HINVAL.VVMA, or for any actions taken in VS-mode, because M-level software is not expected to need to involve itself in VS-stage address translation. For virtual machines, it should be sufficient, and in all likelihood faster as well, to leave VS-stage address translation alone and merge all other translation stages into G-stage shadow page tables controlled by **hgap**. This assumption does place some constraints on possible future RISC-V*

extensions that current machines will be able to emulate efficiently.

The hypervisor extension changes the behavior of the Modify Privilege field, MPRV, of **mstatus**. When MPRV=0, translation and protection behave as normal. When MPRV=1, explicit memory accesses are translated and protected, and endianness is applied, as though the current virtualization mode were set to MPV and the current nominal privilege mode were set to MPP. Table 132 enumerates the cases.

Table 132. Effect of MPRV on the translation and protection of explicit memory accesses.

MPRV	MPV	MPP	Effect
0	-	-	Normal access; current privilege mode applies.
1	0	0	U-level access with HS-level translation and protection only.
1	0	1	HS-level access with HS-level translation and protection only.
1	-	3	M-level access with no translation.
1	1	0	VU-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. vsstatus .MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage.
1	1	1	VS-level access with two-stage translation and protection. The HS-level MXR bit makes any executable page readable. vsstatus .MXR makes readable those pages marked executable at the VS translation stage, but only if readable at the guest-physical translation stage. vsstatus .SUM applies instead of the HS-level SUM bit.

MPRV does not affect the virtual-machine load/store instructions, HLV, HLVX, and HSV. The explicit loads and stores of these instructions always act as though V=1 and the nominal privilege mode were **hstatus**.SPVP, overriding MPRV.

The **mstatus** register is a superset of the HS-level **sstatus** register but is not a superset of **vsstatus**.

5.4.2. Machine Interrupt Delegation (**mideleg**) Register

When the hypervisor extension is implemented, bits 10, 6, and 2 of **mideleg** (corresponding to the standard VS-level interrupts) are each read-only one. Furthermore, if any guest external interrupts are implemented (GEILEN is nonzero), bit 12 of **mideleg** (corresponding to supervisor-level guest external interrupts) is also read-only one. VS-level interrupts and guest external interrupts are always delegated past M-mode to HS-mode.

For bits of **mideleg** that are zero, the corresponding bits in **hideleg**, **hip**, and **hie** are read-only zeros.

5.4.3. Machine Interrupt (**mip** and **mie**) Registers

The hypervisor extension gives registers **mip** and **mie** additional active bits for the hypervisor-added interrupts. Figure 104 and Figure 105 show the standard portions (bits 15:0) of registers **mip** and **mie** when the hypervisor extension is implemented.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIP	SGEIP	MEIP	VSEIP	SEIP	0	MTIP	VSTIP	STIP	0	MSIP	VSSIP	SSIP	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 104. Standard portion (bits 15:0) of **mip**.

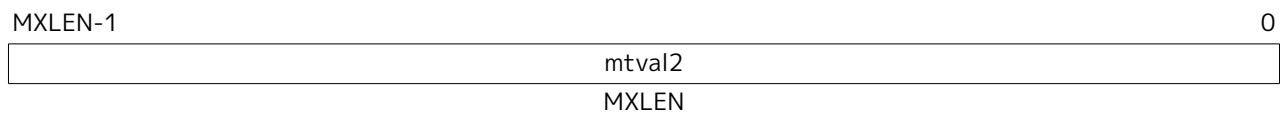
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	LCOFIE	SGEIE	MEIE	VSEIE	SEIE	0	MTIE	VSTIE	STIE	0	MSIE	VSSIE	SSIE	0	0
2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 105. Standard portion (bits 15:0) of **mie**.

Bits SGEIP, VSEIP, VSTIP, and VSSIP in **mip** are aliases for the same bits in hypervisor CSR **hip**, while SGEIE, VSEIE, VSTIE, and VSSIE in **mie** are aliases for the same bits in **hie**.

5.4.4. Machine Second Trap Value (**mtval2**) Register

The **mtval2** register is an MXLEN-bit read/write register formatted as shown in Figure 106. When a trap is taken into M-mode, **mtval2** is written with additional exception-specific information, alongside **mtval**, to assist software in handling the trap.

Figure 106. Machine second trap value register (**mtval2**).

When a guest-page-fault trap is taken into M-mode, **mtval2** is written with either zero or the guest physical address that faulted, shifted right by 2 bits. For other traps, **mtval2** is set to zero, but a future standard or extension may redefine **mtval2**'s setting for other traps.

If a guest-page fault is due to an implicit memory access during first-stage (VS-stage) address translation, a guest physical address written to **mtval2** is that of the implicit memory access that faulted. Additional information is provided in CSR **mtinst** to disambiguate such situations.

Otherwise, for misaligned loads and stores that cause guest-page faults, a nonzero guest physical address in **mtval2** corresponds to the faulting portion of the access as indicated by the virtual address in **mtval**. For instruction guest-page faults on systems with variable-length instructions, a nonzero **mtval2** corresponds to the faulting portion of the instruction as indicated by the virtual address in **mtval**.

mtval2 is a WARL register that must be able to hold zero and may be capable of holding only an arbitrary subset of other 2-bit-shifted guest physical addresses, if any.

The Ssdbltrap extension (See Section 8.10) requires the implementation of the **mtval2** CSR.

5.4.5. Machine Trap Instruction (**mtinst**) Register

The **mtinst** register is an MXLEN-bit read/write register formatted as shown in Figure 107. When a trap is taken into M-mode, **mtinst** is written with a value that, if nonzero, provides information about the instruction that trapped, to assist software in handling the trap. The values that may be written to **mtinst** on a trap are documented in Section 5.6.3.

Figure 107. Machine trap instruction (**mtinst**) register.

mtinst is a WARL register that need only be able to hold the values that the implementation may automatically write to it on a trap.

5.5. Two-Stage Address Translation

Whenever the current virtualization mode V is 1, two-stage address translation and protection is in effect. For any virtual memory access, the original virtual address is converted in the first stage by VS-level address translation, as controlled by the **vsatp** register, into a *guest physical address*. The guest physical address is then converted in the second stage by guest physical address translation, as controlled by the **hgap** register, into a supervisor physical address. The two stages are known also as VS-stage and G-stage translation. Although there is no option to disable two-stage address translation when $V=1$, either stage of translation can be effectively disabled by zeroing the corresponding **vsatp** or **hgap** register.

The **vsstatus** field MXR, which makes execute-only pages readable by explicit loads, only overrides VS-stage page protection. Setting MXR at VS-level does not override guest-physical page protections. Setting MXR at HS-level, however, overrides both VS-stage and G-stage execute-only permissions.

When $V=1$, memory accesses that would normally bypass address translation are subject to G-stage address translation alone. This includes memory accesses made in support of VS-stage address translation, such as reads and writes of VS-level page tables.

Machine-level physical memory protection applies to supervisor physical addresses and is in effect regardless of virtualization mode.

5.5.1. Guest Physical Address Translation

The mapping of guest physical addresses to supervisor physical addresses is controlled by CSR **hgap** (Section 5.2.10).

When the address translation scheme selected by the MODE field of **hgap** is Bare, guest physical addresses are equal to supervisor physical addresses without modification, and no memory protection applies in the trivial translation of guest physical addresses to supervisor physical addresses.

When **hgap**.MODE specifies a translation scheme of Sv32x4, Sv39x4, Sv48x4, or Sv57x4, G-stage address translation is a variation on the usual page-based virtual address translation scheme of Sv32, Sv39, Sv48, or Sv57, respectively. In each case, the size of the incoming address is widened by 2 bits (to 34, 41, 50, or 59 bits). To accommodate the 2 extra bits, the root page table (only) is expanded by a factor of four to be 16 KiB instead of the usual 4 KiB. Matching its larger size, the root page table also must be aligned to a 16 KiB boundary instead of the usual 4 KiB page boundary. Except as noted, all other aspects of Sv32, Sv39, Sv48, or Sv57 are adopted unchanged for G-stage translation. Non-root page tables and all page table entries (PTEs) have the same formats as documented in Section 4.3, Section 4.4, Section 4.5, and Section 4.6.

For Sv32x4, an incoming guest physical address is partitioned into a virtual page number (VPN) and page offset as shown in Figure 108. This partitioning is identical to that for an Sv32 virtual address as depicted in Figure 58, except with 2 more bits at the high end in VPN[1]. (Note that the fields of a partitioned guest physical address also correspond one-for-one with the structure that Sv32 assigns to a physical address, depicted in Figure 58.)

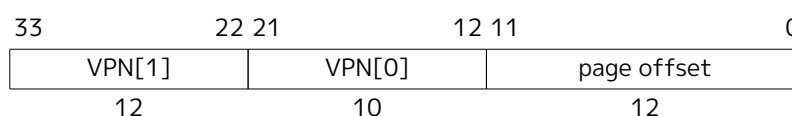


Figure 108. Sv32x4 virtual address (guest physical address).

For Sv39x4, an incoming guest physical address is partitioned as shown in Figure 109. This partitioning is identical to that for an Sv39 virtual address as depicted in Figure 61, except with 2 more bits at the high end in VPN[2]. Address bits 63:41 must all be zeros, or else a guest-page-fault exception occurs.

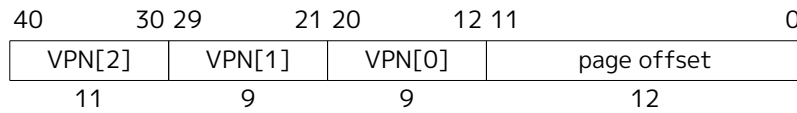


Figure 109. Sv39x4 virtual address (guest physical address).

For Sv48x4, an incoming guest physical address is partitioned as shown in Figure 110. This partitioning is identical to that for an Sv48 virtual address as depicted in Figure 64, except with 2 more bits at the high end in VPN[3]. Address bits 63:50 must all be zeros, or else a guest-page-fault exception occurs.

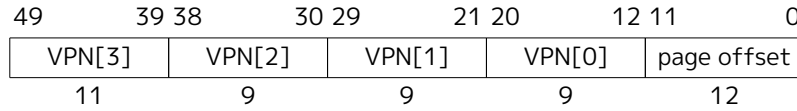


Figure 110. Sv48x4 virtual address (guest physical address).

For Sv57x4, an incoming guest physical address is partitioned as shown in Figure 111. This partitioning is identical to that for an Sv57 virtual address as depicted in Figure 67, except with 2 more bits at the high end in VPN[4]. Address bits 63:59 must all be zeros, or else a guest-page-fault exception occurs.

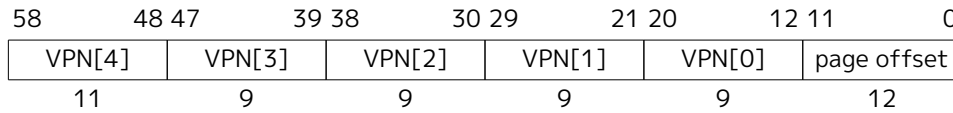


Figure 111. Sv57x4 virtual address (guest physical address).

The page-based G-stage address translation scheme for RV32, Sv32x4, is defined to support a 34-bit guest physical address so that an RV32 hypervisor need not be limited in its ability to virtualize real 32-bit RISC-V machines, even those with 33-bit or 34-bit physical addresses. This may include the possibility of a machine virtualizing itself, if it happens to use 33-bit or 34-bit physical addresses. Multiplying the size and alignment of the root page table by a factor of four is the cheapest way to extend Sv32 to cover a 34-bit address. The possible wastage of 12 KiB for an unnecessarily large root page table is expected to be of negligible consequence for most (maybe all) real uses.

A consistent ability to virtualize machines having as much as four times the physical address space as virtual address space is believed to be of some utility also for RV64. For a machine implementing 39-bit virtual addresses (Sv39), for example, this allows the hypervisor extension to support up to a 41-bit guest physical address space without either necessitating hardware support for 48-bit virtual addresses (Sv48) or falling back to emulating the larger address space using shadow page tables.

The conversion of an Sv32x4, Sv39x4, Sv48x4, or Sv57x4 guest physical address is accomplished with the same algorithm used for Sv32, Sv39, Sv48, or Sv57, as presented in Section 4.3.2, except that:

- **hgap** substitutes for the usual **satp**;
- for the translation to begin, the effective privilege mode must be VS-mode or VU-mode;
- when checking the U bit, the current privilege mode is always taken to be U-mode; and
- guest-page-fault exceptions are raised instead of regular page-fault exceptions.

For G-stage address translation, all memory accesses (including those made to access data structures for VS-stage address translation) are considered to be user-level accesses, as though executed in U-mode. Access type permissions—readable, writable, or executable—are checked during G-stage translation the same as for VS-stage translation. For a memory access made to support VS-stage address translation (such as to read/write a VS-level page table), permissions and the need to set A and/or D bits at the G-stage level

are checked as though for an implicit load or store, not for the original access type. However, any exception is always reported for the original access type (instruction, load, or store/AMO).

The G bit in all G-stage PTEs is currently not used. Until its use is defined by a standard extension, it should be cleared by software for forward compatibility, and must be ignored by hardware.



G-stage address translation uses the identical format for PTEs as regular address translation, even including the U bit, due to the possibility of sharing some (or all) page tables between G-stage translation and regular HS-level address translation. Regardless of whether this usage will ever become common, we chose not to preclude it.

5.5.2. Guest-Page Faults

Guest-page-fault traps may be delegated from M-mode to HS-mode under the control of CSR `medeleg`, but cannot be delegated to other privilege modes. On a guest-page fault, CSR `mtval` or `stval` is written with the faulting guest virtual address as usual, and `mtval2` or `htval` is written either with zero or with the faulting guest physical address, shifted right by 2 bits. CSR `mtinst` or `htinst` may also be written with information about the faulting instruction or other reason for the access, as explained in [Section 5.6.3](#).

When an instruction fetch or a misaligned memory access straddles a page boundary, two different address translations are involved. When a guest-page fault occurs in such a circumstance, the faulting virtual address written to `mtval/stval` is the same as would be required for a regular page fault. Thus, the faulting virtual address may be a page-boundary address that is higher than the instruction's original virtual address, if the byte at that page boundary is among the accessed bytes.

When a guest-page fault is not due to an implicit memory access for VS-stage address translation, a nonzero guest physical address written to `mtval2/htval` shall correspond to the exact virtual address written to `mtval/stval`.

5.5.3. Memory-Management Fences

The behavior of the SFENCE.VMA instruction is affected by the current virtualization mode V. When V=0, the virtual-address argument is an HS-level virtual address, and the ASID argument is an HS-level ASID. The instruction orders stores only to HS-level address-translation structures with subsequent HS-level address translations.

When V=1, the virtual-address argument to SFENCE.VMA is a guest virtual address within the current virtual machine, and the ASID argument is a VS-level ASID within the current virtual machine. The current virtual machine is identified by the VMID field of CSR `hgatp`, and the effective ASID can be considered to be the combination of this VMID with the VS-level ASID. The SFENCE.VMA instruction orders stores only to the VS-level address-translation structures with subsequent VS-stage address translations for the same virtual machine, i.e., only when `hgatp.VMID` is the same as when the SFENCE.VMA executed.

Hypervisor instructions HFENCE.VVMA and HFENCE.GVMA provide additional memory-management fences to complement SFENCE.VMA. These instructions are described in [Section 5.3.2](#).

[Section 3.7.2](#) discusses the intersection between physical memory protection (PMP) and page-based address translation. It is noted there that, when PMP settings are modified in a manner that affects either the physical memory that holds page tables or the physical memory to which page tables point, M-mode software must synchronize the PMP settings with the virtual memory system. For HS-level address translation, this is accomplished by executing in M-mode an SFENCE.VMA instruction with `rs1=x0` and `rs2=x0`, after the PMP CSRs are written. Synchronization with G-stage and VS-stage data structures is also needed. Executing an HFENCE.GVMA instruction with `rs1=x0` and `rs2=x0` suffices to flush all G-stage or

VS-stage address-translation cache entries that have cached PMP settings corresponding to the final translated supervisor physical address. An `HFENCE.VVMA` instruction is not required.

Similarly, if the setting of the PBMTE or ADUE bits in `menvcfg` are changed, an `HFENCE.GVMA` instruction with `rs1=x0` and `rs2=x0` suffices to synchronize with respect to the altered interpretation of G-stage and VS-stage PTEs' PBMT and A/D bit fields, respectively.

By contrast, if the PBMTE or ADUE bits in `henvcfg` are changed, executing an `HFENCE.VVMA` with `rs1=x0` and `rs2=x0` suffices to synchronize with respect to the altered interpretation of VS-stage PTEs' PBMT and A/D bit fields for the currently active VMID.



No mechanism is provided to atomically change `vsatp` and `hgap` together. Hence, to prevent speculative execution causing one guest's VS-stage translations to be cached under another guest's VMID, world-switch code should zero `vsatp`, then swap `hgap`, then finally write the new `vsatp` value. Similarly, if `henvcfg.PBMT/ADUE` need be world-switched, they should be switched after zeroing `vsatp` but before writing the new `vsatp` value, obviating the need to execute an `HFENCE.VVMA` instruction.

5.5.4. Interaction with Pointer Masking

Guest physical addresses (GPAs) are 2 bits wider than the corresponding virtual address translation modes, resulting in additional address translation schemes Sv32x4, Sv39x4, Sv48x4, and Sv57x4 for translating guest physical addresses to supervisor physical addresses. When running with virtualization in VS/VU mode with `vsatp.MODE = Bare`, this means that those two bits may be subject to pointer masking, depending on `hgap.MODE` and `senvcfg.PMM/henvcfg.PMM` (for VU/VS mode). If `vsatp.MODE != BARE`, this issue does not apply.



An implementation could mask those two bits on the TLB access path, but this can have a significant timing impact. Alternatively, an implementation may choose to "waste" TLB capacity by having up to 4 duplicate entries for each page. In this case, the pointer masking operation can be applied on the TLB refill path, where it is unlikely to affect timing. To support this approach, some TLB entries need to be flushed when `PMLen` changes in a way that may affect these duplicate entries.

To support implementations where `(XLEN-PMLen)` can be less than the GPA width supported by `hgap.MODE`, hypervisors should execute an `HFENCE.GVMA` with `rs1=x0` if the `henvcfg.PMM` is changed from or to a value where `(XLEN-PMLen)` is less than GPA width supported by the `hgap` translation mode of that guest. Specifically, these cases are:

- `PMLen=7` and `hgap.MODE=sv57x4`
- `PMLen=16` and `hgap.MODE=sv57x4`
- `PMLen=16` and `hgap.MODE=sv48x4`

Implementation of an address-specific `HFENCE.GVMA` should either ignore the address argument, or should ignore the top masked GPA bits of entries when comparing for an address match.

5.6. Traps

5.6.1. Trap Cause Codes

The hypervisor extension augments the trap cause encoding. [Table 133](#) lists the possible M-mode and HS-mode trap cause codes when the hypervisor extension is implemented. Codes are added for VS-level

interrupts (interrupts 2, 6, 10), for supervisor-level guest external interrupts (interrupt 12), for virtual-instruction exceptions (exception 22), and for guest-page faults (exceptions 20, 21, 23). Furthermore, environment calls from VS-mode are assigned cause 10, whereas those from HS-mode or S-mode use cause 9 as usual.

Table 133. Machine and supervisor cause register (**mcause** and **scause**) values when the hypervisor extension is implemented.

Interrupt	Exception Code	Description
1	0	<i>Reserved</i>
1	1	Supervisor software interrupt
1	2	Virtual supervisor software interrupt
1	3	Machine software interrupt
1	4	<i>Reserved</i>
1	5	Supervisor timer interrupt
1	6	Virtual supervisor timer interrupt
1	7	Machine timer interrupt
1	8	<i>Reserved</i>
1	9	Supervisor external interrupt
1	10	Virtual supervisor external interrupt
1	11	Machine external interrupt
1	12	Supervisor guest external interrupt
1	13	Counter-overflow interrupt
1	14-15	<i>Reserved</i>
1	≥16	<i>Designated for platform use</i>

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode or VU-mode
0	9	Environment call from HS-mode
0	10	Environment call from VS-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16	Double trap
0	17	<i>Reserved</i>
0	18	Software check
0	19	Hardware error
0	20	Instruction guest-page fault
0	21	Load guest-page fault
0	22	Virtual instruction
0	23	Store/AMO guest-page fault
0	24-31	<i>Designated for custom use</i>
0	32-47	<i>Reserved</i>
0	48-63	<i>Designated for custom use</i>
0	≥64	<i>Reserved</i>

HS-mode and VS-mode ECALLs use different cause values so they can be delegated separately.

When $V=1$, a virtual-instruction exception (code 22) is normally raised instead of an illegal-instruction exception if the attempted instruction is *HS-qualified* but is prevented from executing when $V=1$ either due to insufficient privilege or because the instruction is expressly disabled by a supervisor or hypervisor CSR such as `scounteren` or `hcounteren`. An instruction is *HS-qualified* if it would be valid to execute in HS-mode (for some values of the instruction's register operands), assuming fields `TSR` and `TVM` of CSR `mstatus` are both zero.

A special rule applies for CSR instructions that access 32-bit high-half CSRs such as `cycleh` and `htimedeltah`. When $V=1$ and $XLEN=32$, an invalid attempt to access a high-half CSR raises a virtual-instruction exception instead of an illegal-instruction exception if the same CSR instruction for the corresponding low-half CSR (e.g. `cycle` or `htimedelta`) is HS-qualified.



When $XLEN>32$, an attempt to access a high-half CSR always raises an illegal-instruction exception.

Specifically, a virtual-instruction exception is raised for the following cases:

- in VS-mode, attempts to access a non-high-half counter CSR when the corresponding bit in `hcounteren` is 0 and the same bit in `mcounteren` is 1;

- in VS-mode, if XLEN=32, attempts to access a high-half counter CSR when the corresponding bit in **hcounteren** is 0 and the same bit in **mcounteren** is 1;
- in VU-mode, attempts to access a non-high-half counter CSR when the corresponding bit in either **hcounteren** or **scounteren** is 0 and the same bit in **mcounteren** is 1;
- in VU-mode, if XLEN=32, attempts to access a high-half counter CSR when the corresponding bit in either **hcounteren** or **scounteren** is 0 and the same bit in **mcounteren** is 1;
- in VS-mode or VU-mode, attempts to execute a hypervisor instruction (HLV, HLVX, HSV, or HFENCE);
- in VS-mode or VU-mode, attempts to access an implemented non-high-half hypervisor CSR or VS CSR when the same access (read/write) would be allowed in HS-mode, assuming **mstatus.TVM**=0;
- in VS-mode or VU-mode, if XLEN=32, attempts to access an implemented high-half hypervisor CSR or high-half VS CSR when the same access (read/write) to the CSR's low-half partner would be allowed in HS-mode, assuming **mstatus.TVM**=0;
- in VU-mode, attempts to execute WFI when **mstatus.TW**=0, or to execute a supervisor instruction (SRET or SFENCE);
- in VU-mode, attempts to access an implemented non-high-half supervisor CSR when the same access (read/write) would be allowed in HS-mode, assuming **mstatus.TVM**=0;
- in VU-mode, if XLEN=32, attempts to access an implemented high-half supervisor CSR when the same access to the CSR's low-half partner would be allowed in HS-mode, assuming **mstatus.TVM**=0;
- in VS-mode, attempts to execute WFI when **hstatus.VTW**=1 and **mstatus.TW**=0, unless the instruction completes within an implementation-specific, bounded time;
- in VS-mode, attempts to execute SRET when **hstatus.VTSR**=1; and
- in VS-mode, attempts to execute an SFENCE.VMA or SINVAL.VMA instruction or to access **satp**, when **hstatus.VTVM**=1.

Other extensions to the RISC-V Privileged Architecture may add to the set of circumstances that cause a virtual-instruction exception when V=1.

On a virtual-instruction trap, **mtval** or **stval** is written the same as for an illegal-instruction trap.



It is not unusual that hypervisors must emulate the instructions that raise virtual-instruction exceptions, to support nested hypervisors or for other reasons. Machine level is expected ordinarily to delegate virtual-instruction traps directly to HS-level, whereas illegal-instruction traps are likely to be processed first in M-mode before being conditionally delegated (by software) to HS-level. Consequently, virtual-instruction traps are expected typically to be handled faster than illegal-instruction traps.

When not emulating the trapping instruction, a hypervisor should convert a virtual-instruction trap into an illegal-instruction exception for the guest virtual machine.

*Because TSR and TVM in **mstatus** are intended to impact only S-mode (HS-mode), they are ignored for determining exceptions in VS-mode.*

Fields FS and VS in registers **sstatus** and **vsstatus** deviate from the usual HS-qualified rule. If an instruction is prevented from executing because FS or VS is zero in either **sstatus** or **vsstatus**, the exception raised is always an illegal-instruction exception, never a virtual-instruction exception.



*Early implementations of the H extension treated FS and VS in **sstatus** and **vsstatus** specially this way, and the behavior has been codified to maintain compatibility for software.*

Table 134. Synchronous exception priority when the hypervisor extension is implemented.

Priority	Exc.Code	Description
Highest	3	Instruction address breakpoint
	12, 20, 1	During instruction address translation: First encountered page fault, guest-page fault, or access fault
	1	With physical address for instruction: Instruction access fault
	2 22 0 8, 9, 10, 11 3 3	Illegal instruction Virtual instruction Instruction address misaligned Environment call Environment break Load/store/AMO address breakpoint
	4,6	Optionally: Load/store/AMO address misaligned
	13, 15, 21, 23, 5, 7	During address translation for an explicit memory access: First encountered page fault, guest-page fault, or access fault
	5, 7	With physical address for an explicit memory access: Load/store/AMO access fault
Lowest	4, 6	If not higher priority: Load/store/AMO address misaligned

If an instruction may raise multiple synchronous exceptions, the decreasing priority order of Table 134 indicates which exception is taken and reported in `mcause` or `scause`.

5.6.2. Trap Entry

When a trap occurs in HS-mode or U-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode. When a trap occurs in VS-mode or VU-mode, it goes to M-mode, unless delegated by `medeleg` or `mideleg`, in which case it goes to HS-mode, unless further delegated by `hedeleg` or `hideleg`, in which case it goes to VS-mode.

When a trap is taken into M-mode, virtualization mode `V` gets set to 0, and fields `MPV` and `MPP` in `mstatus` (or `mstatush`) are set according to Table 135. A trap into M-mode also writes fields `GVA`, `MPiE`, and `MiE` in `mstatus/mstatush` and writes CSRs `mepc`, `mcause`, `mtval`, `mtval2`, and `mtinst`.

Table 135. Value of `mstatus/mstatush` fields `MPV` and `MPP` after a trap into M-mode. Upon trap return, `MPV` is ignored when `MPP=3`.

Previous Mode	MPV	MPP
U-mode	0	0
HS-mode	0	1
M-mode	0	3
VU-mode	1	0
VS-mode	1	1

When a trap is taken into HS-mode, virtualization mode `V` is set to 0, and `hstatus.SPv` and `sstatus.SPv` are set according to Table 136. If `V` was 1 before the trap, field `SPVP` in `hstatus` is set the same as `sstatus.SPv`; otherwise, `SPVP` is left unchanged. A trap into HS-mode also writes field `GVA` in `hstatus`, fields `SPiE` and `SiE` in `sstatus`, and CSRs `sepc`, `scause`, `stval`, `htval`, and `htinst`.

Table 136. Value of **hstatus** field SPV and **sstatus** field SPP after a trap into HS-mode.

Previous Mode	SPV	SPP
U-mode	0	0
HS-mode	0	1
VU-mode	1	0
VS-mode	1	1

When a trap is taken into VS-mode, **vsstatus**.SPP is set according to Table 137. Register **hstatus** and the HS-level **sstatus** are not modified, and the virtualization mode V remains 1. A trap into VS-mode also writes fields SPIE and SIE in **vsstatus** and writes CSRs **vsepc**, **vscause**, and **vtval**.

Table 137. Value of **vsstatus** field SPP after a trap into VS-mode.

Previous Mode	SPP
VU-mode	0
VS-mode	1

5.6.3. Transformed Instruction or Pseudoinstruction for **mtinst** or **htinst**

On any trap into M-mode or HS-mode, one of these values is written automatically into the appropriate trap instruction CSR, **mtinst** or **htinst**:

- zero;
- a transformation of the trapping instruction;
- a custom value (allowed only if the trapping instruction is non-standard); or
- a special pseudoinstruction.

Except when a pseudoinstruction value is required (described later), the value written to **mtinst** or **htinst** may always be zero, indicating that the hardware is providing no information in the register for this particular trap.



The value written to the trap instruction CSR serves two purposes. The first is to improve the speed of instruction emulation in a trap handler, partly by allowing the handler to skip loading the trapping instruction from memory, and partly by obviating some of the work of decoding and executing the instruction. The second purpose is to supply, via pseudoinstructions, additional information about guest-page-fault exceptions caused by implicit memory accesses done for VS-stage address translation.

A transformation of the trapping instruction is written instead of simply a copy of the original instruction in order to minimize the burden for hardware yet still provide to a trap handler the information needed to emulate the instruction. An implementation may at any time reduce its effort by substituting zero in place of the transformed instruction.

On an interrupt, the value written to the trap instruction register is always zero. On a synchronous exception, if a nonzero value is written, one of the following shall be true about the value:

- Bit 0 is **1**, and replacing bit 1 with **1** makes the value into a valid encoding of a standard instruction.

In this case, the instruction that trapped is the same kind as indicated by the register value, and the register value is the transformation of the trapping instruction, as defined later. For example, if bits 1:0 are binary **11** and the register value is the encoding of a standard LW (load word) instruction, then the trapping instruction is LW, and the register value is the transformation of the trapping LW instruction.

- Bit 0 is **1**, and replacing bit 1 with **1** makes the value into an instruction encoding that is explicitly designated for a custom instruction (*not* an unused reserved encoding).

This is a *custom value*. The instruction that trapped is a non-standard instruction. The interpretation of a custom value is not otherwise specified by this standard.

- The value is one of the special pseudoinstructions defined later, all of which have bits 1:0 equal to **00**.

These three cases exclude a large number of other possible values, such as all those having bits 1:0 equal to binary **10**. A future standard or extension may define additional cases, thus allowing values that are currently excluded. Software may safely treat an unrecognized value in a trap instruction register the same as zero.



*To be forward-compatible with future revisions of this standard, software that interprets a nonzero value from **mtinst** or **htinst** must fully verify that the value conforms to one of the cases listed above. For instance, for RV64, discovering that bits 6:0 of **mtinst** are **0000011** and bits 14:12 are **010** is not sufficient to establish that the first case applies and the trapping instruction is a standard LW instruction; rather, software must also confirm that bits 63:32 of **mtinst** are all zeros. A future standard might define new values for 64-bit **mtinst** that are nonzero in bits 63:32 yet may coincidentally have in bits 31:0 the same bit patterns as standard RV64 instructions.*

Unlike for standard instructions, there is no requirement that the instruction encoding of a custom value be of the same “kind” as the instruction that trapped (or even have any correlation with the trapping instruction).

Table 138 shows the values that may be automatically written to the trap instruction register for each standard exception cause. For exceptions that prevent the fetching of an instruction, only zero or a pseudoinstruction value may be written. A custom value may be automatically written only if the instruction that traps is non-standard. A future standard or extension may permit other values to be written, chosen from the set of allowed values established earlier.

Table 138. Values that may be automatically written to the trap instruction (**mtinst** or **htinst**) register on an exception trap.

Exception	Zero	Transformed Standard Instruction	Custom Value	Pseudoinstruction Value
Instruction address misaligned	Yes	No	Yes	No
Instruction access fault	Yes	No	No	No
Illegal instruction	Yes	No	No	No
Breakpoint	Yes	No	Yes	No
Virtual instruction	Yes	No	Yes	No
Load address misaligned	Yes	Yes	Yes	No
Load access fault	Yes	Yes	Yes	No
Store/AMO address misaligned	Yes	Yes	Yes	No
Store/AMO access fault	Yes	Yes	Yes	No
Environment call	Yes	No	Yes	No
Instruction page fault	Yes	No	No	No
Load page fault	Yes	Yes	Yes	No
Store/AMO page fault	Yes	Yes	Yes	No
Instruction guest-page fault	Yes	No	No	Yes
Load guest-page fault	Yes	Yes	Yes	Yes
Store/AMO guest-page fault	Yes	Yes	Yes	Yes

As enumerated in the table, a synchronous exception may write to the trap instruction register a standard transformation of the trapping instruction only for exceptions that arise from explicit memory accesses (from loads, stores, and AMO instructions). Accordingly, standard transformations are currently defined only for these memory-access instructions. If a synchronous trap occurs for a standard instruction for which no transformation has been defined, the trap instruction register shall be written with zero (or, under certain circumstances, with a special pseudoinstruction value).

For a standard load instruction that is not a compressed instruction and is one of LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH, the transformed instruction has the format shown in Figure 112.

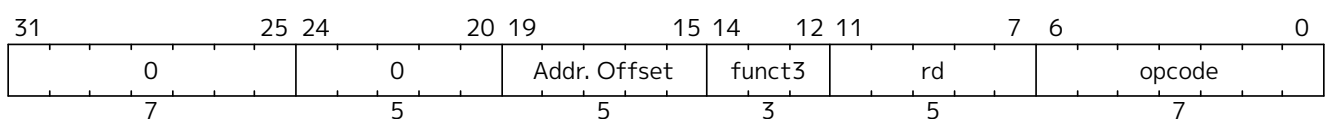


Figure 112. Transformed load instruction (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH). Fields **funct3**, **rd**, and **opcode** are the same as the trapping load instruction.

For a standard store instruction that is not a compressed instruction and is one of SB, SH, SW, SD, FSW, FSD, FSQ, or FSH, the transformed instruction has the format shown in Figure 113.

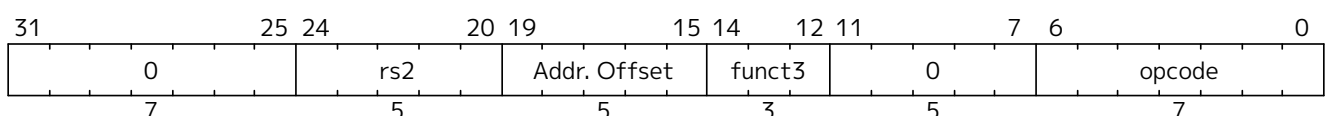


Figure 113. Transformed store instruction (SB, SH, SW, SD, FSW, FSD, FSQ, or FSH). Fields **rs2**, **funct3**, and **opcode** are the same as the trapping store instruction.

For a standard atomic instruction (load-reserved, store-conditional, or AMO instruction), the transformed instruction has the format shown in Figure 114.

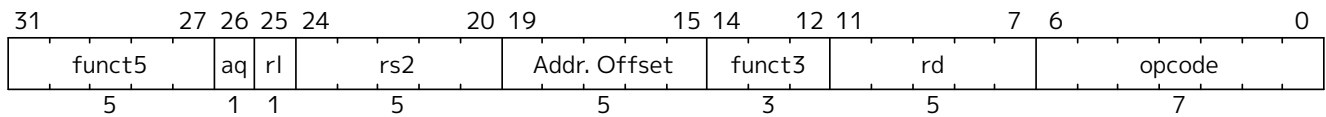


Figure 114. Transformed atomic instruction (load-reserved, store-conditional, or AMO instruction). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset.

For a standard virtual-machine load/store instruction (HLV, HLVX, or HSV), the transformed instruction has the format shown in Figure 115.

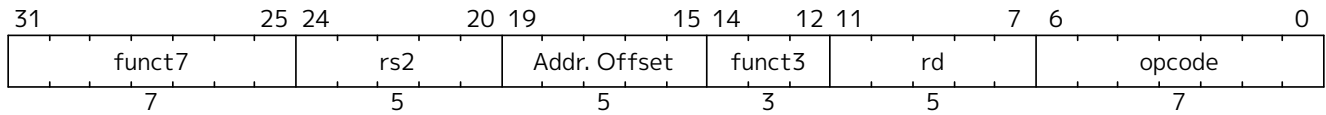


Figure 115. Transformed virtual-machine load/store instruction (HLV, HLVX, HSV). All fields are the same as the trapping instruction except bits 19:15, Addr. Offset

In all the transformed instructions above, the Addr. Offset field that replaces the instruction's rs1 field in bits 19:15 is the positive difference between the faulting virtual address (written to **mtval** or **stval**) and the original virtual address. This difference can be nonzero only for a misaligned memory access. Note also that, for basic loads and stores, the transformations replace the instruction's immediate offset fields with zero.

For a standard compressed instruction (16-bit size), the transformed instruction is found as follows:

1. Expand the compressed instruction to its 32-bit equivalent.
2. Transform the 32-bit equivalent instruction.
3. Replace bit 1 with a 0.

Bits 1:0 of a transformed standard instruction will be binary **01** if the trapping instruction is compressed and **11** if not.



*In decoding the contents of **mtinst** or **htinst**, once software has determined that the register contains the encoding of a standard basic load (LB, LBU, LH, LHU, LW, LWU, LD, FLW, FLD, FLQ, or FLH) or basic store (SB, SH, SW, SD, FSW, FSD, FSQ, or FSH), it is not necessary to confirm also that the immediate offset fields (31:25, and 24:20 or 11:7) are zeros. The knowledge that the register's value is the encoding of a basic load/store is sufficient to prove that the trapping instruction is of the same kind.*

A future version of this standard may add information to the fields that are currently zeros. However, for backwards compatibility, any such information will be for performance purposes only and can safely be ignored.

For guest-page faults, the trap instruction register is written with a special pseudoinstruction value if: (a) the fault is caused by an implicit memory access for VS-stage address translation, and (b) a nonzero value (the faulting guest physical address) is written to **mtval2** or **htval1**. If both conditions are met, the value written to **mtinst** or **htinst** must be taken from Table 139; zero is not allowed.

Table 139. Special pseudoinstruction values for guest-page faults. The RV32 values are used when VSXLEN=32, and the RV64 values when VSXLEN=64.

Value	Meaning
0x00002000	32-bit read for VS-stage address translation (RV32)
0x00002020	32-bit write for VS-stage address translation (RV32)

Value	Meaning
0x00003000	64-bit read for VS-stage address translation (RV64)
0x00003020	64-bit write for VS-stage address translation (RV64)

The defined pseudoinstruction values are designed to correspond closely with the encodings of basic loads and stores, as illustrated by [Table 140](#).

Table 140. Standard instructions corresponding to the special pseudoinstructions of [Table 139](#).

Encoding	Instruction
0x00002003	lw x0,0(x0)
0x00002023	sw x0,0(x0)
0x00003003	ld x0,0(x0)
0x00003023	sd x0,0(x0)

A *write* pseudoinstruction (0x00002020 or 0x00003020) is used for the case that the machine is attempting automatically to update bits A and/or D in VS-level page tables. All other implicit memory accesses for VS-stage address translation will be reads. If a machine never automatically updates bits A or D in VS-level page tables (leaving this to software), the *write* case will never arise. The fact that such a page table update must actually be atomic, not just a simple write, is ignored for the pseudoinstruction.

*If the conditions that necessitate a pseudoinstruction value can ever occur for M-mode, then **mtinst** cannot be entirely read-only zero; and likewise for HS-mode and **htinst**. However, in that case, the trap instruction registers may minimally support only values 0 and 0x00002000 or 0x00003000, and possibly 0x00002020 or 0x00003020, requiring as few as one or two flip-flops in hardware, per register.*

There is no harm here in ignoring the atomicity requirement for page table updates, because a hypervisor is not expected in these circumstances to emulate an implicit memory access that fails. Rather, the hypervisor is given enough information about the faulting access to be able to make the memory accessible (e.g. by restoring a missing page of virtual memory) before resuming execution by retrying the faulting instruction.

5.6.4. Trap Return

The MRET instruction is used to return from a trap taken into M-mode. MRET first determines what the new privilege mode will be according to the values of MPP and MPV in **mstatus** or **mstatush**, as encoded in [Table 135](#). MRET then in **mstatus**/**mstatush** sets MPV=0, MPP=0, MIE=MPIE, and MPIE=1. Lastly, MRET sets the privilege mode as previously determined, and sets **pc=mepc**.

The SRET instruction is used to return from a trap taken into HS-mode or VS-mode. Its behavior depends on the current virtualization mode.

When executed in M-mode or HS-mode (i.e., V=0), SRET first determines what the new privilege mode will be according to the values in **hstatus.SPV** and **sstatus.SPP**, as encoded in [Table 136](#). SRET then sets **hstatus.SPV**=0, and in **sstatus** sets SPP=0, SIE=SPIE, and SPIE=1. Lastly, SRET sets the privilege mode as previously determined, and sets **pc=sepc**.

When executed in VS-mode (i.e., V=1), SRET sets the privilege mode according to [Table 137](#), in **vsstatus** sets SPP=0, SIE=SPIE, and SPIE=1, and lastly sets **pc=vsepc**.

If the Ssdbltrap extension is implemented, when SRET is executed in HS-mode, if the new privilege mode is

VU, the **SRET** instruction sets **vsstatus.SDT** to 0. When executed in VS-mode, **vsstatus.SDT** is set to 0.

DRAFT

Chapter 6. "Sm" Machine Extensions



This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

6.1. "Smstateen/Ssstateen" Extensions, Version 1.0

The implementation of optional RISC-V extensions has the potential to open covert channels between separate user threads, or between separate guest OSes running under a hypervisor. The problem occurs when an extension adds processor state — usually explicit registers, but possibly other forms of state — that the main OS or hypervisor is unaware of (and hence won't context-switch) but that can be modified/written by one user thread or guest OS and perceived/examined/read by another.

For example, the Advanced Interrupt Architecture (AIA) for RISC-V adds to a hart as many as ten supervisor-level CSRs (**siselect**, **sireg**, **stopi**, **sseteipnum**, **sclreipnum**, **sseteienum**, **sclreienum**, **sclaimei**, **sieh**, and **siph**) and provides also the option for hardware to be backward-compatible with older, pre-AIA software. Because an older hypervisor that is oblivious to the AIA will not know to swap any of the AIA's new CSRs on context switches, the registers may then be used as a covert channel between multiple guest OSes that run atop this hypervisor. Although traditional practices might consider such a communication channel harmless, the intense focus on security today argues that a means be offered to plug such channels.

The **f** registers of the RISC-V floating-point extensions and the **v** registers of the vector extension would similarly be potential covert channels between user threads, except for the existence of the FS and VS fields in the **sstatus** register. Even if an OS is unaware of, say, the vector extension and its **v** registers, access to those registers is blocked when the VS field is initialized to zero, either at machine level or by the OS itself initializing **sstatus**.

Obviously, one way to prevent the use of new user-level CSRs as covert channels would be to add to **mstatus** or **sstatus** an "XS" field for each relevant extension, paralleling the V extension's VS field. However, this is not considered a general solution to the problem due to the number of potential future extensions that may add small amounts of state. Even with a 64-bit **sstatus** (necessitating adding **sstatush** for RV32), it is not certain there are enough remaining bits in **sstatus** to accommodate all future user-level extensions. In any event, there is no need to strain **sstatus** (and add **sstatush**) for this purpose. The "enable" flags that are needed to plug covert channels are not generally expected to require swapping on context switches of user threads, making them a less-than-compelling candidate for inclusion in **sstatus**. Hence, a new place is provided for them instead.

6.1.1. State Enable Extensions

The Smstateen and Ssstateen extensions collectively specify machine-mode and supervisor-mode features. The Smstateen extension specification comprises the **mstateen***, **sstateen***, and **hstateen*** CSRs and their functionality. The Ssstateen extension specification comprises only the **sstateen*** and **hstateen*** CSRs and their functionality.

For RV64 harts, this extension adds four new 64-bit CSRs at machine level: **mstateen0** (Machine State Enable 0), **mstateen1**, **mstateen2**, and **mstateen3**.

If supervisor mode is implemented, another four CSRs are defined at supervisor level: **sstateen0**, **sstateen1**, **sstateen2**, and **sstateen3**.

And if the hypervisor extension is implemented, another set of CSRs is added: **hstateen0**, **hstateen1**, **hstateen2**, and **hstateen3**.

For RV32, there are CSR addresses for accessing the upper 32 bits of corresponding machine-level and hypervisor CSRs: **mstateen0h**, **mstateen1h**, **mstateen2h**, **mstateen3h**, **hstateen0h**, **hstateen1h**, **hstateen2h**, and **hstateen3h**.

For the supervisor-level **sstateen** registers, high-half CSRs are not added at this time because it is expected the upper 32 bits of these registers will always be zeros, as explained later below.

Each bit of a **stateen** CSR controls less-privileged access to an extension's state, for an extension that was not deemed "worthy" of a full XS field in **sstatus** like the FS and VS fields for the F and V extensions. The number of registers provided at each level is four because it is believed that $4 * 64 = 256$ bits for machine and hypervisor levels, and $4 * 32 = 128$ bits for supervisor level, will be adequate for many years to come, perhaps for as long as the RISC-V ISA is in use. The exact number four is an attempted compromise between providing too few bits on the one hand and going overboard with CSRs that will never be used on the other. A possible future doubling of the number of **stateen** CSRs is covered later.

The **stateen** registers at each level control access to state at all less-privileged levels, but not at its own level. This is analogous to how the existing **counteren** CSRs control access to performance counter registers. Just as with the **counteren** CSRs, when a **stateen** CSR prevents access to state by less-privileged levels, an attempt in one of those privilege modes to execute an instruction that would read or write the protected state raises an illegal-instruction exception, or, if executing in VS or VU mode and the circumstances for a virtual-instruction exception apply, raises a virtual-instruction exception instead of an illegal-instruction exception.

When this extension is not implemented, all state added by an extension is accessible as defined by that extension.

When a **stateen** CSR prevents access to state for a privilege mode, attempting to execute in that privilege mode an instruction that *implicitly* updates the state without reading it may or may not raise an illegal-instruction or virtual-instruction exception. Such cases must be disambiguated by being explicitly specified one way or the other.

In some cases, the bits of the **stateen** CSRs will have a dual purpose as enables for the ISA extensions that introduce the controlled state.

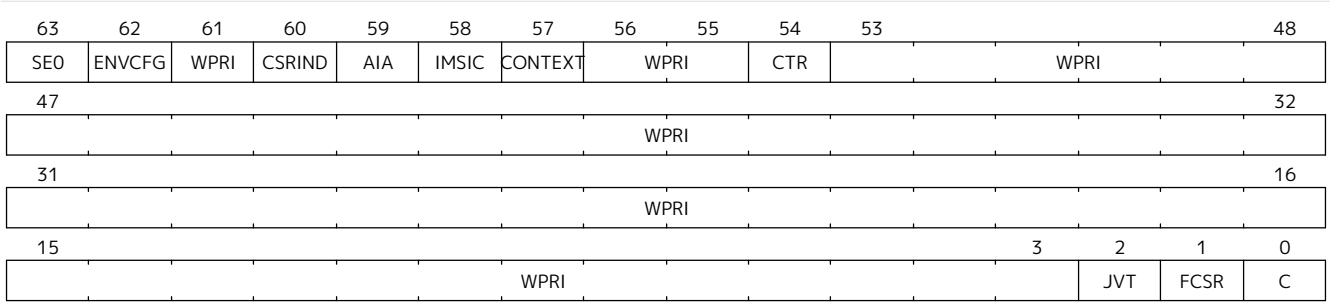
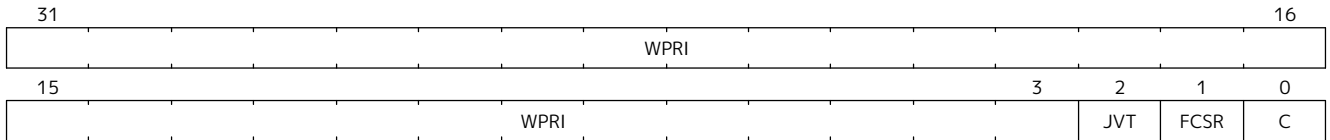
Each bit of a supervisor-level **sstateen** CSR controls user-level access (from U-mode or VU-mode) to an extension's state. The intention is to allocate the bits of **sstateen** CSRs starting at the least-significant end, bit 0, through to bit 31, and then on to the next-higher-numbered **sstateen** CSR.

For every bit with a defined purpose in an **sstateen** CSR, the same bit is defined in the matching **mstateen** CSR to control access below machine level to the same state. The upper 32 bits of an **mstateen** CSR (or for RV32, the corresponding high-half CSR) control access to state that is inherently inaccessible to user level, so no corresponding enable bits in the supervisor-level **sstateen** CSR are applicable. The intention is to allocate bits for this purpose starting at the most-significant end, bit 63, through to bit 32, and then on to the next-higher **mstateen** CSR. If the rate that bits are being allocated from the least-significant end for **sstateen** CSRs is sufficiently low, allocation from the most-significant end of **mstateen** CSRs may be allowed to encroach on the lower 32 bits before jumping to the next-higher **mstateen** CSR. In that case, the bit positions of "encroaching" bits will remain forever read-only zeros in the matching **sstateen** CSRs.

With the hypervisor extension, the **hstateen** CSRs have identical encodings to the **mstateen** CSRs, except controlling accesses for a virtual machine (from VS and VU modes).

Each standard-defined bit of a **stateen** CSR is WARL and may be read-only zero or one, subject to the following conditions.

Bits in any **stateen** CSR that are defined to control state that a hart doesn't implement are read-only zeros

Figure 117. Hypervisor State Enable O Register (**hstateen0**)Figure 118. Supervisor State Enable O Register (**sstateen0**)

The C bit controls access to any and all custom state. The C bit of these registers is not custom state itself; it is a standard field of a standard CSR, either **mstateen0**, **hstateen0**, or **sstateen0**.



The requirements that non-standard extensions must meet to be conforming are not relaxed due solely to changes in the value of this bit. In particular, if software sets this bit but does not execute any custom instructions or access any custom state, the software must continue to execute as specified by all relevant RISC-V standards, or the hardware is not standard-conforming.

The FCSR bit controls access to **fcsr** for the case when floating-point instructions operate on **x** registers instead of **f** registers as specified by the Zfinx and related extensions (Zdinx, etc.). Whenever **misa.F** = 1, FCSR bit of **mstateen0** is read-only zero (and hence read-only zero in **hstateen0** and **sstateen0** too). For convenience, when the **stateen** CSRs are implemented and **misa.F** = 0, then if the FCSR bit of a controlling **stateen0** CSR is zero, all floating-point instructions cause an illegal-instruction exception (or virtual-instruction exception, if relevant), as though they all access **fcsr**, regardless of whether they really do.

The JVT bit controls access to the **jvt** CSR provided by the Zcmt extension.

The SEO bit in **mstateen0** controls access to the **hstateen0**, **hstateen0h**, and the **sstateen0** CSRs. The SEO bit in **hstateen0** controls access to the **sstateen0** CSR.

The ENVCFG bit in **mstateen0** controls access to the **henvcfg**, **henvcfgh**, and the **senvcfg** CSRs. The ENVCFG bit in **hstateen0** controls access to the **senvcfg** CSRs.

The CSRIND bit in **mstateen0** controls access to the **siselect**, **sireg***, **vsiselect**, and the **vsireg*** CSRs provided by the Sscsrind extensions. The CSRIND bit in **hstateen0** controls access to the **siselect** and the **sireg***, (really **vsiselect** and **vsireg***) CSRs provided by the Sscsrind extensions.

The IMSIC bit in **mstateen0** controls access to the IMSIC state, including CSRs **stopei** and **vstopei**, provided by the Ssaia extension. The IMSIC bit in **hstateen0** controls access to the guest IMSIC state, including CSRs **stopei** (really **vstopei**), provided by the Ssaia extension.



*Setting the IMSIC bit in **hstateen0** to zero prevents a virtual machine from accessing the hart's IMSIC the same as setting **hstatus.VGEIN** = 0.*

The AIA bit in **mstateen0** controls access to all state introduced by the Ssaia extension and not controlled by either the CSRIND or the IMSIC bits. The AIA bit in **hstateen0** controls access to all state introduced by the Ssaia extension and not controlled by either the CSRIND or the IMSIC bits of **hstateen0**.

The **CONTEXT** bit in **mstateen0** controls access to the **scontext** and **hcontext** CSRs provided by the **Sdtrig** extension. The **CONTEXT** bit in **hstateen0** controls access to the **scontext** CSR provided by the **Sdtrig** extension.

The **P1P13** bit in **mstateen0** controls access to the **hedeleg** introduced by Privileged Specification Version 1.13.

The **SRMCFG** bit in **mstateen0** controls access to the **srmcfg** CSR introduced by the **Ssqosid** [Section 8.1](#) extension.

6.1.3. Usage

After the writable bits of the machine-level **mstateen** CSRs are initialized to zeros on reset, machine-level software can set bits in these registers to enable less-privileged access to the controlled state. This may be either because machine-level software knows how to swap the state or, more likely, because machine-level software isn't swapping supervisor-level environments. (Recall that the main reason the **mstateen** CSRs must exist is so machine level can emulate the hypervisor extension. When machine level isn't emulating the hypervisor extension, it is likely there will be no need to keep any implemented **mstateen** bits zero.)

If machine level sets any writable **mstateen** bits to nonzero, it must initialize the matching **hstateen** CSRs, if they exist, by writing zeros to them. And if any **mstateen** bits that are set to one have matching bits in the **sstateen** CSRs, machine-level software must also initialize those **sstateen** CSRs by writing zeros to them. Ordinarily, machine-level software will want to set bit 63 of all **mstateen** CSRs, necessitating that it write zero to all **hstateen** CSRs.

Software should ensure that all writable bits of **sstateen** CSRs are initialized to zeros when an OS at supervisor level is first entered. The OS can then set bits in these registers to enable user-level access to the controlled state, presumably because it knows how to context-swap the state.

For the **sstateen** CSRs whose access by a guest OS is permitted by bit 63 of the corresponding **hstateen** CSRs, a hypervisor must include the **sstateen** CSRs in the context it swaps for a guest OS. When it starts a new guest OS, it must ensure the writable bits of those **sstateen** CSRs are initialized to zeros, and it must emulate accesses to any other **sstateen** CSRs.

If software at any privilege level does not support multiple contexts for less-privilege levels, then it may choose to maximize less-privileged access to all state by writing a value of all ones to the **stateen** CSRs at its level (the **mstateen** CSRs for machine level, the **sstateen** CSRs for an OS, and the **hstateen** CSRs for a hypervisor), without knowing all the state to which it is granting access. This is justified because there is no risk of a covert channel between execution contexts at the less-privileged level when only one context exists at that level. This situation is expected to be common for machine level, and it might also arise, for example, for a type-1 hypervisor that hosts only a single guest virtual machine.

*If a need is anticipated, the set of **stateen** CSRs could in the future be doubled by adding these:*



- 0x38C **mstateen4**, 0x39C **mstateen4h**
- 0x38D **mstateen5**, 0x39D **mstateen5h**
- 0x38E **mstateen6**, 0x39E **mstateen6h**
- 0x38F **mstateen7**, 0x39F **mstateen7h**
- 0x18C **sstateen4**
- 0x18D **sstateen5**
- 0x18E **sstateen6**

- 0x18F sstateen7
- 0x68C hstateen4, 0x69C hstateen4h
- 0x68D hstateen5, 0x69D hstateen5h
- 0x68E hstateen6, 0x69E hstateen6h
- 0x68F hstateen7, 0x69F hstateen7h

These additional CSRs are not a definite part of the original proposal because it is unclear whether they will ever be needed, and it is believed the rate of consumption of bits in the first group, registers numbered 0-3, will be slow enough that any looming shortage will be perceptible many years in advance. At the moment, it is not known even how many years it may take to exhaust just mstateen0, sstateen0, and hstateen0.

6.2. "Smcsrind/Sscsrind" Indirect CSR Access, Version 1.0

6.2.1. Introduction

Smcsrind/Sscsrind is an ISA extension that extends the indirect CSR access mechanism originally defined as part of the [Smaia/Ssaia extensions](#), in order to make it available for use by other extensions without creating an unnecessary dependence on Smaia/Ssaia.

This extension confers two benefits:

1. It provides a means to access an array of registers via CSRs without requiring allocation of large chunks of the limited CSR address space.
2. It enables software to access each of an array of registers by index, without requiring a switch statement with a case for each register.

CSRs are accessed indirectly via this extension using select values, in contrast to being accessed directly using standard CSR numbers. A CSR accessible via one method may or may not be accessible via the other method. Select values are a separate address space from CSR numbers, and from tselect values in the Sdtrig extension. If a CSR is both directly and indirectly accessible, the CSR's select value is unrelated to its CSR number.

Further, Machine-level and Supervisor-level select values are separate address spaces from each other; however, Machine-level and Supervisor-level CSRs with the same select value may be defined by an extension as partial or full aliases with respect to each other. This typically would be done for CSRs that can be delegated from Machine-level to Supervisor-level.

The machine-level extension **Smcsrind** encompasses all added CSRs and all behavior modifications for a hart, over all privilege levels. For a supervisor-level environment, extension **Sscsrind** is essentially the same as Smcsrind except excluding the machine-level CSRs and behavior not directly visible to supervisor level.

6.2.2. Machine-level CSRs

Number	Privilege	Width	Name	Description
0x350	MRW	XLEN	miselect	Machine indirect register select
0x351	MRW	XLEN	mireg	Machine indirect register alias
0x352	MRW	XLEN	mireg2	Machine indirect register alias 2

Number	Privilege	Width	Name	Description
0x353	MRW	XLEN	mireg3	Machine indirect register alias 3
0x355	MRW	XLEN	mireg4	Machine indirect register alias 4
0x356	MRW	XLEN	mireg5	Machine indirect register alias 5
0x357	MRW	XLEN	mireg6	Machine indirect register alias 6



*The **mireg*** CSR numbers are not consecutive because **miph** is CSR number 0x354.*

The CSRs listed in the table above provide a window for accessing register state indirectly. The value of **miselect** determines which register is accessed upon read or write of each of the machine indirect alias CSRs (**mireg***). **miselect** value ranges are allocated to dependent extensions, which specify the register state accessible via each **miregi** register, for each **miselect** value. **miselect** is a WARL register.

The **miselect** register implements at least enough bits to support all implemented **miselect** values (corresponding to the implemented extensions that utilize **miselect/mireg*** to indirectly access register state). The **miselect** register may be read-only zero if there are no extensions implemented that utilize it.

Values of **miselect** with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of **miselect** with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of **miselect** moves to the new position, retaining its value from before.



*An implementation is not required to support any custom values for **miselect**.*

The behavior upon accessing **mireg*** from M-mode, while **miselect** holds a value that is not implemented, is UNSPECIFIED.



It is expected that implementations will typically raise an illegal-instruction exception for such accesses, so that, for example, they can be identified as software bugs. Platform specs, profile specs, and/or the Privileged ISA spec may place more restrictions on behavior for such accesses.

Attempts to access **mireg*** while **miselect** holds a number in an allocated and implemented range results in a specific behavior that, for each combination of **miselect** and **miregi**, is defined by the extension to which the **miselect** value is allocated.



*Ordinarily, each **miregi** will access register state, access read-only 0 state, or raise an illegal-instruction exception.*

*For RV32, if an extension defines an indirectly accessed register as 64 bits wide, it is recommended that the lower 32 bits of the register are accessed through one of **mireg**, **mireg2**, or **mireg3**, while the upper 32 bits are accessed through **mireg4**, **mireg5**, or **mireg6**, respectively.*



*Six ***ireg*** registers are defined in order to ensure that the needs of extensions in development are covered, with some room for growth. For example, for an **siselect** value associated with counter X, **sireg/sireg2** could be used to access **mhpcounterX/mhpmeventX**, while **sireg4/sireg5** could access **mhpcounterXh/mhpmeventXh**. Six ***ireg*** registers allows for accessing up to 3 CSR arrays per index (***iselect**) with RV32-only CSRs, or up to 6 CSR arrays per index value without RV32-only CSRs.*

6.2.3. Supervisor-level CSRs

Number	Privilege	Width	Name	Description
0x150	SRW	XLEN	siselect	Supervisor indirect register select
0x151	SRW	XLEN	sireg	Supervisor indirect register alias
0x152	SRW	XLEN	sireg2	Supervisor indirect register alias 2
0x153	SRW	XLEN	sireg3	Supervisor indirect register alias 3
0x155	SRW	XLEN	sireg4	Supervisor indirect register alias 4
0x156	SRW	XLEN	sireg5	Supervisor indirect register alias 5
0x157	SRW	XLEN	sireg6	Supervisor indirect register alias 6

The CSRs in the table above are required if S-mode is implemented.

The **siselect** register will support the value range 0..0xFFFF at a minimum. A future extension may define a value range outside of this minimum range. Only if such an extension is implemented will **siselect** be required to support larger values.



*Requiring a range of 0–0xFFFF for **siselect**, even though most or all of the space may be reserved or inaccessible, permits M-mode to emulate indirectly accessed registers in this implemented range, including registers that may be standardized in the future.*

Values of **siselect** with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of **siselect** with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of **siselect** moves to the new position, retaining its value from before.

The behavior upon accessing **sireg*** from M-mode or S-mode, while **siselect** holds a value that is not implemented at supervisor level, is UNSPECIFIED.



It is recommended that implementations raise an illegal-instruction exception for such accesses, to facilitate possible emulation (by M-mode) of these accesses.



*An extension is considered not to be implemented at supervisor level if machine level has disabled the extension for S-mode, such as by the settings of certain fields in CSR **menvcfg**, for example.*

Otherwise, attempts to access **sireg*** from M-mode or S-mode while **siselect** holds a number in a standard-defined and implemented range result in specific behavior that, for each combination of **siselect** and **siregi**, is defined by the extension to which the **siselect** value is allocated.



*Ordinarily, each **siregi** will access register state, access read-only 0 state, or, unless executing in a virtual machine (covered in the next section), raise an illegal-instruction exception.*

Note that the widths of **siselect** and **sireg*** are always the current XLEN rather than SXLEN. Hence, for example, if MXLEN = 64 and SXLEN = 32, then these registers are 64 bits when the current privilege mode is M (running RV64 code) but 32 bits when the privilege mode is S (RV32 code).

6.2.4. Virtual Supervisor-level CSRs

Number	Privilege	Width	Name	Description
0x250	HRW	XLEN	vsiselect	Virtual supervisor indirect register select
0x251	HRW	XLEN	vsireg	Virtual supervisor indirect register alias
0x252	HRW	XLEN	vsireg2	Virtual supervisor indirect register alias 2
0x253	HRW	XLEN	vsireg3	Virtual supervisor indirect register alias 3
0x255	HRW	XLEN	vsireg4	Virtual supervisor indirect register alias 4
0x256	HRW	XLEN	vsireg5	Virtual supervisor indirect register alias 5
0x257	HRW	XLEN	vsireg6	Virtual supervisor indirect register alias 6

The CSRs in the table above are required if the hypervisor extension is implemented. These VS CSRs all match supervisor CSRs, and substitute for those supervisor CSRs when executing in a virtual machine (in VS-mode or VU-mode).

The **vsiselect** register will support the value range 0..0xFFFF at a minimum. A future extension may define a value range outside of this minimum range. Only if such an extension is implemented will **vsiselect** be required to support larger values.



*Requiring a range of 0–0xFFFF for **vsiselect**, even though most or all of the space may be reserved or inaccessible, permits a hypervisor to emulate indirectly accessed registers in this implemented range, including registers that may be standardized in the future.*

*More generally it is recommended that **vsiselect** and **siselect** be implemented with the same number of bits. This also avoids creation of a virtualization hole due to observable differences between **vsiselect** and **siselect** widths.*

Values of **vsiselect** with the most-significant bit set (bit XLEN - 1 = 1) are designated only for custom use, presumably for accessing custom registers through the alias CSRs. Values of **vsiselect** with the most-significant bit clear are designated only for standard use and are reserved until allocated to a standard architecture extension. If XLEN is changed, the most-significant bit of **vsiselect** moves to the new position, retaining its value from before.

For alias CSRs **sireg*** and **vsireg***, the hypervisor extension's usual rules for when to raise a virtual-instruction exception (based on whether an instruction is HS-qualified) are not applicable. The rules given in this section for **sireg** and **vsireg** apply instead, unless overridden by the requirements specified in the section below, which take precedence over this section when extension Smstateen is also implemented.

A virtual-instruction exception is raised for attempts from VS-mode or VU-mode to directly access **vsiselect** or **vsireg***, or attempts from VU-mode to access **siselect** or **sireg***.

The behavior upon accessing **vsireg*** from M-mode or HS-mode, or accessing **sireg*** (really **vsireg***) from VS-mode, while **vsiselect** holds a value that is not implemented at HS level, is UNSPECIFIED.



It is recommended that implementations raise an illegal-instruction exception for such accesses, to facilitate possible emulation (by M-mode) of these accesses.

Otherwise, while **vsiselect** holds a number in a standard-defined and implemented range, attempts to access **vsireg*** from a sufficiently privileged mode, or to access **sireg*** (really **vsireg***) from VS-mode, result in specific behavior that, for each combination of **vsiselect** and **vsiregi**, is defined by the extension to which the **vsiselect** value is allocated.



Ordinarily, each **vsiregi** will access register state, access read-only *O* state, or raise an exception (either an illegal-instruction exception or, for select accesses from VS-mode, a virtual-instruction exception). When **vsiselect** holds a value that is implemented at HS level but not at VS level, attempts to access **sireg*** (really **vsireg***) from VS-mode will typically raise a virtual-instruction exception. But there may be cases specific to an extension where different behavior is more appropriate.

Like **siselect** and **sireg***, the widths of **vsiselect** and **vsireg*** are always the current XLEN rather than VSXLEN. Hence, for example, if HSXLEN = 64 and VSXLEN = 32, then these registers are 64 bits when accessed by a hypervisor in HS-mode (running RV64 code) but 32 bits for a guest OS in VS-mode (RV32 code).

6.2.5. Access control by the state-enable CSRs

If extension Smstateen is implemented together with Smcsrind, bit 60 of state-enable register **mstateen0** controls access to **siselect**, **sireg***, **vsiselect**, and **vsireg***. When **mstateen0**[60]=0, an attempt to access one of these CSRs from a privilege mode less privileged than M-mode results in an illegal-instruction exception. As always, the state-enable CSRs do not affect the accessibility of any state when in M-mode, only in less privileged modes. For more explanation, see the documentation for extension Smstateen in [Section 6.1](#).

Other extensions may specify that certain mstateen bits control access to registers accessed indirectly through **siselect** + **sireg***, and/or **vsiselect** + **vsireg***. However, regardless of any other mstateen bits, if **mstateen0**[60] = 1, a virtual-instruction exception is raised as described in the previous section for all attempts from VS-mode or VU-mode to directly access **vsiselect** or **vsireg***, and for all attempts from VU-mode to access **siselect** or **sireg***.

If the hypervisor extension is implemented, the same bit is defined also in hypervisor CSR **hstateen0**, but controls access to only **siselect** and **sireg*** (really **vsiselect** and **vsireg***), which is the state potentially accessible to a virtual machine executing in VS or VU-mode. When **hstateen0**[60]=0 and **mstateen0**[60]=1, all attempts from VS or VU-mode to access **siselect** or **sireg*** raise a virtual-instruction exception, not an illegal-instruction exception, regardless of the value of **vsiselect** or any other mstateen bit.

Extension Ssstateen is defined as the supervisor-level view of Smstateen. Therefore, the combination of Sscsrind and Ssstateen incorporates the bit defined above for **hstateen0** but not that for **mstateen0**, since machine-level CSRs are not visible to supervisor level.



CSR address space is reserved for a possible future "Sscsrind" extension that extends indirect CSR access to user mode.

6.3. "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode, Version 1.0

Being able to access the memory of a process running at a high privileged execution mode, such as the Supervisor or Machine mode, from a lower privileged mode such as the User mode, introduces an obvious attack vector since it allows for an attacker to perform privilege escalation, and tamper with the code and/or data of that process. A less obvious attack vector exists when the reverse happens, in which case an attacker instead of tampering with code and/or data that belong to a high-privileged process, can tamper with the memory of an unprivileged / less-privileged process and trick the high-privileged process to use or execute it.

Two mechanisms combine to prevent this attack vector. The first one prevents the OS from accessing the memory of an unprivileged process unless a specific code path is followed, and the second one prevents the

OS from executing the memory of an unprivileged process at all times. RISC-V already includes support for the former through the `sstatus.SUM` bit, and for the latter by always denying supervisor execution of virtual memory pages marked with the U bit.

Terms:



- **PMP Entry:** A pair of `pmppcfg[i]` / `pmppaddr[i]` registers.
- **PMP Rule:** The contents of a `pmppcfg` register and its associated `pmppaddr` register(s), that encode a valid protected physical memory region, where `pmppcfg[i].A != OFF`, and if `pmppcfg[i].A == TOR`, `pmppaddr[i-1] < pmppaddr[i]`.
- **Ignored:** Any permissions set by a matching PMP rule are ignored, and all accesses to the requested address range are allowed.
- **Enforced:** Only access types configured in the PMP rule matching the requested address range are allowed; failures will cause an access-fault exception.
- **Denied:** Any permissions set by a matching PMP rule are ignored, and no accesses to the requested address range are allowed.; failures will cause an access-fault exception.
- **Locked:** A PMP rule/entry where the `pmppcfg.L` bit is set.
- **PMP reset:** A reset process where all PMP settings of the hart, including locked rules/settings, are re-initialized to a set of safe defaults, before releasing the hart (back) to the firmware / OS / application.

6.3.1. Threat model

The rationale that guided development of this extension is included in Section [Appendix A.1](#).

Without the Smepmp extension, it is not possible for a PMP rule to be **enforced** only on non-Machine modes and **denied** on Machine mode, in order to allow access to a memory region solely by less-privileged modes. It is only possible to have a **locked** rule that will be **enforced** on all modes, or a rule that will be **enforced** on non-Machine modes and be **ignored** by Machine mode. So for any physical memory region which is not protected with a Locked rule, Machine mode has unlimited access, including the ability to execute it.

Without being able to protect less-privileged modes from Machine mode, it is not possible to prevent the mentioned attack vector. This becomes even more important for RISC-V than on other architectures, since implementations are allowed where a hart only has Machine and User modes available, so the whole OS will run on Machine mode instead of the non-existent Supervisor mode. In such implementations the attack surface is greatly increased, and the same kind of attacks performed on Supervisor mode and mitigated through the virtual-memory system, can be performed on Machine mode without any available mitigations. Even on implementations with Supervisor mode present attacks are still possible against the Firmware and/or the Secure Monitor running on Machine mode.

6.3.2. Smepmp Physical Memory Protection Rules

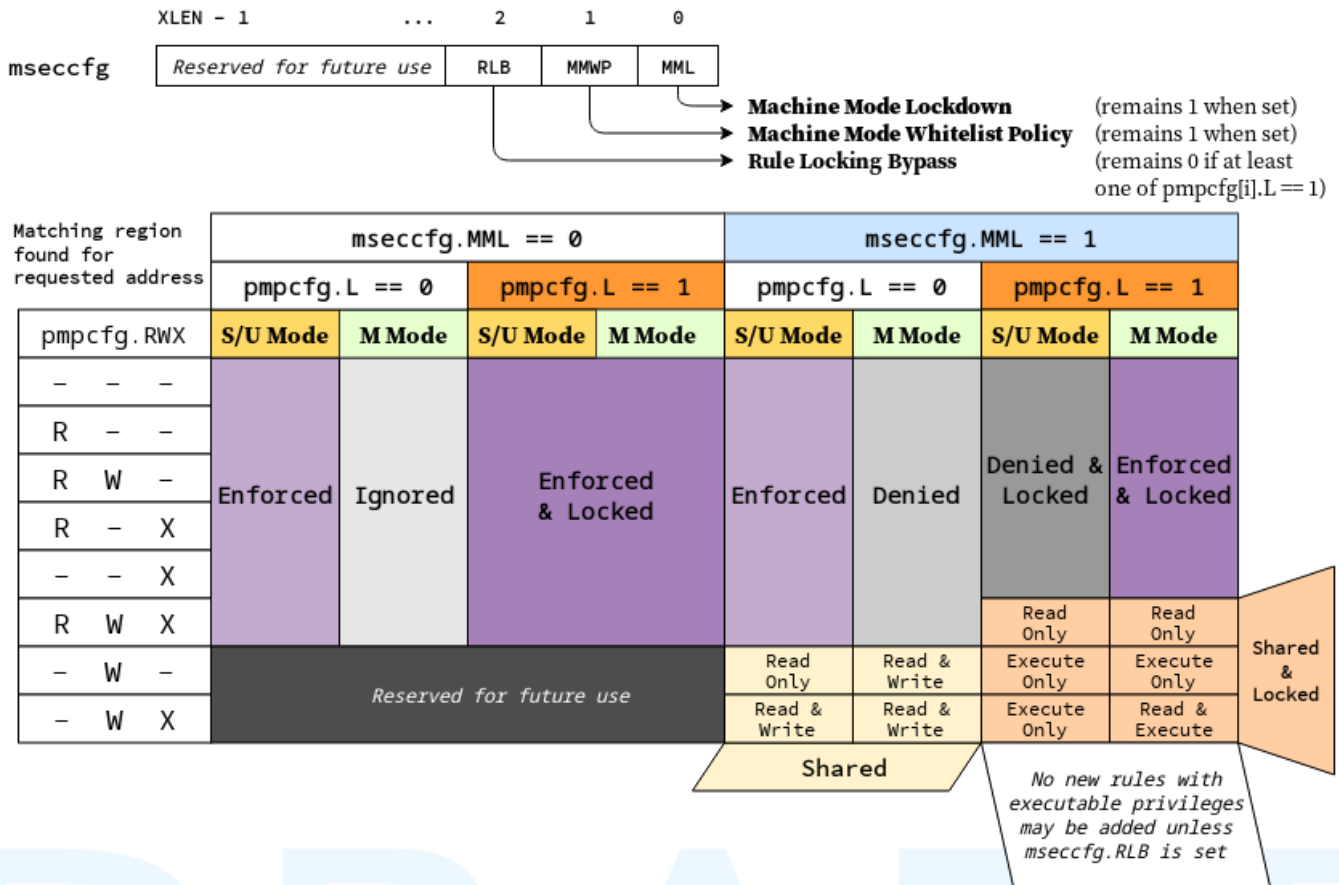
To address the threat model outlined in Section [Section 6.3.1](#), this extension introduces the **RLB**, **MMWP**, and **MML** fields in the `mseccfg` CSR and their associated rules. See [Figure 33](#) for the detailed specification of these fields and the corresponding rules.

The physical memory protection rules when `mseccfg.MML` is set to 1 are summarized in the truth table below.

Bits on <i>pmpcfg</i> register				Result	
L	R	W	X	M Mode	S/U Mode
0	0	0	0	Inaccessible region (Access Exception)	
0	0	0	1	Access Exception	Execute-only region
0	0	1	0	Shared data region: Read/write on M mode, read-only on S/U mode	
0	0	1	1	Shared data region: Read/write for both M and S/U mode	
0	1	0	0	Access Exception	Read-only region
0	1	0	1	Access Exception	Read/Execute region
0	1	1	0	Access Exception	Read/Write region
0	1	1	1	Access Exception	Read/Write/Execute region
1	0	0	0	Locked inaccessible region* (Access Exception)	
1	0	0	1	Locked Execute-only region*	Access Exception
1	0	1	0	Locked Shared code region: Execute only on both M and S/U mode.*	
1	0	1	1	Locked Shared code region: Execute only on S/U mode, read/execute on M mode.*	
1	1	0	0	Locked Read-only region*	Access Exception
1	1	0	1	Locked Read/Execute region*	Access Exception
1	1	1	0	Locked Read/Write region*	Access Exception
1	1	1	1	Locked Shared data region: Read only on both M and S/U mode.*	

: *Locked rules cannot be removed or modified until a **PMP reset**, unless **mseccfg.RLB** is set.

A visual representation of these rules is as follows:



6.3.3. Smepmp software discovery

Since all fields defined in `mseccfg` as part of this extension are locked when set (`MMWP/MML`) or locked when cleared (`RLB`), software can't poll them for determining the presence of Smepmp. It is expected that BootROM will set `mseccfg.MMWP` and/or `mseccfg.MML` during early boot, before jumping to the firmware, so that the firmware will be able to determine the presence of Smepmp by reading `mseccfg` and checking the state of `mseccfg.MMWP` and `mseccfg.MML`.

6.4. "Smcntrpmf" Cycle and Instret Privilege Mode Filtering, Version 1.0

6.4.1. Introduction

The cycle and instret counters serve to support user mode self-profiling usages, wherein a user can read the counter(s) twice and compute the delta(s) to evaluate user software performance and behavior. By default, these counters are not filtered by privilege mode, and thus they continue to increment while traps (e.g., page faults or interrupts) to more privileged code are handled. This causes two problems:

- It introduces unpredictable noise to the counter values observed by the user.
- It leaks information about privileged software execution to user mode.

Smcntrpmf remedies these issues by introducing privilege mode filtering for the cycle and instret counters.

6.4.2. CSRs

6.4.2.1. Machine Counter Configuration (mccyclecfg, minstretcfg) Registers

mccyclecfg and minstretcfg are 64-bit registers that configure privilege mode filtering for the cycle and instret counters, respectively.

63	62	61	60	59	58	57:0
0	MINH	SINH	UINH	VSINH	VUINH	WPRI

Field	Description
MINH	If set, then counting of events in M-mode is inhibited
SINH	If set, then counting of events in S/HS-mode is inhibited
UINH	If set, then counting of events in U-mode is inhibited
VSINH	If set, then counting of events in VS-mode is inhibited
VUINH	If set, then counting of events in VU-mode is inhibited

When all xINH bits are zero, event counting is enabled in all modes.

For each bit in 61:58, if the associated privilege mode is not implemented, the bit is read-only zero.

For RV32, bits 63:32 of mccyclecfg can be accessed via the mccyclecfgh CSR, and bits 63:32 of minstretcfg can be accessed via the minstretcfgh CSR.

The content of these registers may be accessible from Supervisor level if the Smcdeleg/Ssccfg extensions are implemented.

The more natural CSR number for mccyclecfg would be 0x320, but that was allocated to mcountinhibit.

This register format matches that specified for programmable counters by Sscofpmf. The bit position for the OF bit (bit 63) is read-only 0, since these counters do not generate local-counter-overflow interrupts on overflow.

6.4.3. Counter Behavior

The fundamental behavior of cycle and instret is modified in that counting does not occur while executing in an inhibited privilege mode. Further, the following defines how transitions between a non-inhibited privilege mode and an inhibited privilege mode are counted.

The cycle counter will simply count CPU cycles while the CPU is in a non-inhibited privilege mode. Mode transition operations (traps and trap returns) may take multiple clock cycles, and the change of privilege mode may be reported as occurring in any one of those cycles (possibly different for each occurrence of a trap or trap return).

The RISC-V ISA has no requirement that the number of cycles for a trap or trap return be the same for all occurrences. Implementations are free to determine the extent to which this number may be consistent and predictable (or not), and the same is true for the specific cycle in which privilege mode changes.

For the instret counter, most instructions do not affect mode transitions, so for those the behavior is clear: instructions that retire in a non-inhibited mode increment instret, and instructions that retire in an

inhibited mode do not. There are two types of instructions that can affect a privilege mode change: instructions that cause synchronous exceptions to a more privileged mode, and `xRET` instructions that return to a less privileged mode. The former are not considered to retire, and hence do not increment `instret`. The latter do retire, and should increment `instret` only if the originating privilege mode is not inhibited.



The `instret` definition above is intended to ensure that the counter increments in a predictable fashion. For example, consider a scenario where `minstretcfg` is configured such that all modes other than U-mode are inhibited. A user mode load should increment only once, even if it takes a page fault or other exception. With this definition, the faulting execution of the load will not increment (it does not retire), the handler instructions will not increment (they execute in an inhibited mode), including the `xRET` (it arguably retires in a non-inhibited mode, but it originates in an inhibited mode). Only once the load is re-executed and retires will it increment `instret`.

In cases where an instruction is emulated by software running in a privilege mode that is inhibited in `minstretcfg`, the emulation routine must emulate the `instret` increment.

6.5. "Smrnmi" Extension for Resumable Non-Maskable Interrupts, Version 1.0

The base machine-level architecture supports only unresumable non-maskable interrupts (UNMIs), where the NMI jumps to a handler in machine mode, overwriting the current `mepc` and `mcause` register values. If the hart had been executing machine-mode code in a trap handler, the previous values in `mepc` and `mcause` would not be recoverable and so execution is not generally resumable.

The Smrnmi extension adds support for resumable non-maskable interrupts (RNMIs) to RISC-V. The extension adds four new CSRs (`mnepc`, `mncause`, `mnstatus`, and `mnscratch`) to hold the interrupted state, and one new instruction, `MNRET`, to resume from the RNMI handler.

6.5.1. RNMI Interrupt Signals

The `rnmi` interrupt signals are inputs to the hart. These interrupts have higher priority than any other interrupt or exception on the hart and cannot be disabled by software. Specifically, they are not disabled by clearing the `mstatus.MIE` register.

6.5.2. RNMI Handler Addresses

The RNMI interrupt trap handler address is implementation-defined.

RNMI also has an associated exception trap handler address, which is implementation defined.



For example, some implementations might use the address specified in `mtvec` as the RNMI exception trap handler.

6.5.3. RNMI CSRs

This extension adds additional M-mode CSRs to enable a resumable non-maskable interrupt (RNMI).



Figure 119. Resumable NMI scratch register `mnscratch`

The **mnscratch** CSR holds an MXLEN-bit read-write register which enables the RNMI trap handler to save and restore the context that was interrupted.

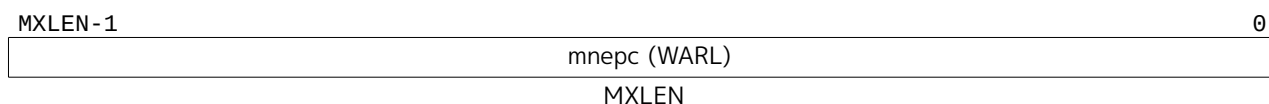


Figure 120. Resumable NMI program counter **mnepc**.

The **mnepc** CSR is an MXLEN-bit read-write register which on entry to the RNMI trap handler holds the PC of the instruction that took the interrupt.

The low bit of **mnepc** (**mnepc[0]**) is always zero. On implementations that support only IALIGN=32, the two low bits (**mnepc[1:0]**) are always zero.

If an implementation allows IALIGN to be either 16 or 32 (by changing CSR **misal**, for example), then, whenever IALIGN=32, bit **mnepc[1]** is masked on reads so that it appears to be 0. This masking occurs also for the implicit read by the MNRET instruction. Though masked, **mnepc[1]** remains writable when IALIGN=32.

mnepc is a **WARL** register that must be able to hold all valid virtual addresses. It need not be capable of holding all possible invalid addresses. Prior to writing **mnepc**, implementations may convert an invalid address into some other invalid address that **mnepc** is capable of holding.



Figure 121. Resumable NMI cause **mncause**.

The **mncause** CSR holds the reason for the RNMI. If the reason is an interrupt, bit MXLEN-1 is set to 1, and the RNMI cause is encoded in the least-significant bits. If the reason is an interrupt and RNMI causes are not supported, bit MXLEN-1 is set to 1, and zero is written to the least-significant bits. If the reason is an exception within M-mode that results in a double trap as specified in the Smdbltrp extension, bit MXLEN-1 is set to 0 and the least-significant bits are set to the cause code corresponding to the exception that precipitated the double trap.

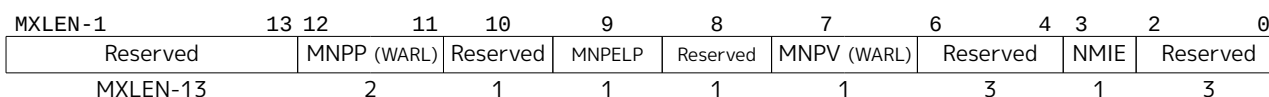


Figure 122. Resumable NMI status register **mnstatus**.

The **mnstatus** CSR holds a two-bit field, MNPP, which on entry to the RNMI trap handler holds the privilege mode of the interrupted context, encoded in the same manner as **mstatus.MPP**. It also holds a one-bit field, MNPV, which on entry to the RNMI trap handler holds the virtualization mode of the interrupted context, encoded in the same manner as **mstatus.MPV**.

If the Zicfilp extension is implemented, **mnstatus** also holds the MNPELP field, which on entry to the RNMI trap handler holds the previous **ELP** state. When an RNMI trap is taken, MNPELP is set to **ELP** and **ELP** is set to 0.

mnstatus also holds the NMIE bit. When NMIE=1, non-maskable interrupts are enabled. When NMIE=0, all interrupts are disabled.

When NMIE=0, the hart behaves as though **mstatus.MPRV** were clear, regardless of the current setting of **mstatus.MPRV**.

Upon reset, NMIE contains the value 0.



RNMIIs are masked out of reset to give software the opportunity to initialize data structures and devices for subsequent RNMI handling.

Software can set NMIE to 1, but attempts to clear NMIE have no effect.

Normally, only reset sequences will explicitly set the NMIE bit.



That the NMIE bit is settable does not suffice to support the nesting of RNMIIs. To support this feature in a direct manner would have required allowing software to clear the NMIE bit—a design choice that would have contravened the concept of non-maskability.

Software that wishes to minimize the latency until the next RNMI is taken can follow the top-half/bottom-half model, where the RNMI handler itself only enqueues a task to a task queue then returns. The bulk of the interrupt servicing is performed later, with RNMIIs enabled.

For the purposes of the WFI instruction, NMIE is a global interrupt enable, meaning that the setting of NMIE does not affect the operation of the WFI instruction.

The other bits in **mnstatus** are *reserved*; software should write zeros and hardware implementations should return zeros.

6.5.4. MNRET Instruction

MNRET is an M-mode-only instruction that uses the values in **mnepc** and **mnstatus** to return to the program counter, privilege mode, and virtualization mode of the interrupted context. This instruction also sets **mnstatus.NMIE**. If MNRET changes the privilege mode to a mode less privileged than M, it also sets **mnstatus.MPRV** to 0. If the Zicfilp extension is implemented, then if the new privileged mode is y, MNRET sets ELP to the logical AND of yLPE (see [Section 6.9.1.1](#)) and **mnstatus.MNPELP**.

6.5.5. RNMI Operation

When an RNMI interrupt is detected, the interrupted PC is written to the **mnepc** CSR, the type of RNMI to the **mncause** CSR, and the privilege mode of the interrupted context to the **mnstatus** CSR. The **mnstatus.NMIE** bit is cleared, masking all interrupts.

The hart then enters machine-mode and jumps to the RNMI trap handler address.

The RNMI handler can resume original execution using the new MNRET instruction, which restores the PC from **mnepc**, the privilege mode from **mnstatus**, and also sets **mnstatus.NMIE**, which re-enables interrupts.

If the hart encounters an exception while executing in M-mode with the **mnstatus.NMIE** bit clear, the actions taken are the same as if the exception had occurred while **mnstatus.NMIE** were set, except that the program counter is set to the RNMI exception trap handler address.



*The Smrnmi extension does not change the behavior of the MRET and SRET instructions. In particular, MRET and SRET are unaffected by the **mnstatus.NMIE** bit, and their execution does not alter the **mnstatus.NMIE** bit.*

6.6. "Smcdeleg/Ssccfg" Counter Delegation Extensions, Version 1.0

In modern “Rich OS” environments, hardware performance monitoring resources are managed by the

kernel, kernel driver, and/or hypervisor. Counters may be configured with differing scopes, in some cases counting events system-wide, while in others counting events on behalf of a single virtual machine or application. In such environments, the latency of counter writes has a direct impact on overall profiling overhead as a result of frequent counter writes during:

1. Sample collection, to clear overflow indication, and reload overflowed counter(s)
2. Context switch, between processes, threads, containers, or virtual machines

These extensions provide a means for M-mode to allow writing select counters and event selectors from S/HS-mode. The purpose is to avert transitions to and from M-mode that add latency to these performance critical supervisor/hypervisor code sections. These extensions also defines one new CSR, `scountinhibit`.

For a Machine-level environment, extension **Smcdeleg** ('Sm' for Privileged architecture and Machine-level extension, 'cdeleg' for Counter Delegation) encompasses all added CSRs and all behavior modifications for a hart, over all privilege levels. For a Supervisor-level environment, extension **Ssccfg** ('Ss' for Privileged architecture and Supervisor-level extension, 'ccfg' for Counter Configuration) provides access to delegated counters, and to new supervisor-level state. For a RISC-V hardware platform, **Smcdeleg** and **Ssccfg** must always be implemented in tandem.

6.6.1. Counter Delegation

The `mcounteren` register allows M-mode to provide the next-lower privilege mode with read access to select counters. When the **Smcdeleg/Ssccfg** extensions are enabled (`menvcfg.CDE=1`), it further allows M-mode to delegate select counters to S-mode.

The `siselect` (and `vsiselect`) index range 0x40-0x5F is reserved for delegated counter access. When a counter i is delegated (`mcounteren[i]=1` and `menvcfg.CDE=1`), the register state associated with counter i can be read or written via `sireg*`, while `siselect` holds $0x40+i$. The counter state accessible via alias CSRs is shown in the table below.

Table 141. Indirect HPM State Mappings

siselect value	sireg	sireg4	sireg2	sireg5
0x40	<code>cycle</code> ¹	<code>cycleh</code> ¹	<code>cyclecfg</code> ¹⁴	<code>cyclecfgh</code> ¹⁴
0x41	<i>See below</i>			
0x42	<code>instret</code> ¹	<code>instreth</code> ¹	<code>instretcfg</code> ¹⁴	<code>instretcfgh</code> ¹⁴
0x43	<code>hpmcounter3</code> ²	<code>hpmcounter3h</code> ²	<code>hpmevent3</code> ²	<code>hpmevent3h</code> ²³
...	...	...	...	...
0x5F	<code>hpmcounter31</code> ²	<code>hpmcounter31h</code> ²	<code>hpmevent31</code> ²	<code>hpmevent31h</code> ²³

¹ Depends on `Zicntr` support

² Depends on `Zihpm` support

³ Depends on `Sscofpmf` support

⁴ Depends on `Smcntrpmf` support

`hpmeventi` may represent a subset of the state accessed by the `mhpmeventi` register. Specifically, if `Sscofpmf` is implemented, event selector bit 62 (MINH) is read-only 0 when accessed through `sireg*`.

Likewise, `cyclecfg` and `instretcfg` may represent a subset of the state accessed by the `mcyclecfg` and `minstretcfg` registers, respectively. If `Smcntrpmf` is implemented, counter configuration register bit 62 (MINH) is read-only 0 when accessed through `sireg*`.

If extension `Smstateen` is implemented, refer to extensions `Smcsrind`/`Sscsrind` (Section 6.2) for how setting bit 60 of CSR `mstateen0` to zero prevents access to registers `siselect`, `sireg*`, `vsiselect`, and `vsireg*` from privileged modes less privileged than M-mode, and likewise how setting bit 60 of `hstateen0` to zero prevents access to `siselect` and `sireg*` (really `vsiselect` and `vsireg*`) from VS-mode.

The remaining rules of this section apply only when access to a CSR is not blocked by `mstateen0[60] = 0` or `hstateen0[60] = 0`.

While the privilege mode is M or S and `siselect` holds a value in the range 0x40-0x5F, illegal-instruction exceptions are raised for the following cases:

- attempts to access any `sireg*` when `menvcfg.CDE = 0`;
- attempts to access `sireg3` or `sireg6`;
- attempts to access `sireg4` or `sireg5` when `XLEN = 64`;
- attempts to access `sireg*` when `siselect = 0x41`, or when the counter selected by `siselect` is not delegated to S-mode (the corresponding bit in `mcounteren` = 0).



The memory-mapped `mtime` register is not a performance monitoring counter to be managed by supervisor software, hence the special treatment of `siselect` value 0x41 described above.

For each `siselect` and `sireg*` combination defined in Table 141, the table further indicates the extensions upon which the underlying counter state depends. If any extension upon which the underlying state depends is not implemented, an attempt from M or S mode to access the given state through `sireg*` raises an illegal-instruction exception.

If the hypervisor (H) extension is also implemented, then as specified by extensions `Smcsrind`/`Sscsrind`, a virtual-instruction exception is raised for attempts from VS-mode or VU-mode to directly access `vsiselect` or `vsireg*`, or attempts from VU-mode to access `siselect` or `sireg*`. Furthermore, while `vsiselect` holds a value in the range 0x40-0x5F:

- An attempt to access any `vsireg*` from M or S mode raises an illegal-instruction exception.
- An attempt from VS-mode to access any `sireg*` (really `vsireg*`) raises an illegal-instruction exception if `menvcfg.CDE = 0`, or a virtual-instruction exception if `menvcfg.CDE = 1`.

6.6.2. Supervisor Counter Inhibit (`scountinhibit`) Register

`Smcdeleg`/`Ssccfg` defines a new `scountinhibit` register, a masked alias of `mcountinhibit`. For counters delegated to S-mode, the associated `mcountinhibit` bits can be accessed via `scountinhibit`. For counters not delegated to S-mode, the associated bits in `scountinhibit` are read-only zero.

When `menvcfg.CDE=0`, attempts to access `scountinhibit` raise an illegal-instruction exception. When Supervisor Counter Delegation is enabled, attempts to access `scountinhibit` from VS-mode or VU-mode raise a virtual-instruction exception.

6.6.3. Virtualizing `scountovf`

For implementations that support `Smcdeleg`/`Ssccfg`, `Sscofpmf`, and the H extension, when `menvcfg.CDE=1`, attempts to read `scountovf` from VS-mode or VU-mode raise a virtual-instruction exception.

6.6.4. Virtualizing Local-Counter-Overflow Interrupts

For implementations that support `Smcdeleg`, `Sscofpmf`, and `Smaia`, the local-counter-overflow interrupt

(LCOFI) bit (bit 13) in each of CSRs **mvip** and **mvien** is implemented and writable.

For implementations that support **Smcdeleg**/**Ssccfg**, **Sscofpmf**, **Smaia**/**Ssaia**, and the H extension, the LCOFI bit (bit 13) in each of **hvip** and **hvien** is implemented and writable.

*The **hvip** register is defined by the hypervisor (H) extension, while the **mvip**, **mvien** and **hvien** registers are defined by the **Smaia**/**Ssaia** extensions.*

*By virtue of implementing **hvip.LCOFI**, it is implicit that the LCOFI bit (bit 13) in each of **vsie** and **vsip** is also implemented.*

*Requiring support for the LCOFI bits listed above ensures that virtual LCOFIs can be delivered to an OS running in S-mode, and to a guest OS running in VS-mode. It is optional whether the LCOFI bit (bit 13) in each of **mideleg** and **hideleg**, which allows all LCOFIs to be delegated to S-mode and VS-mode, respectively, is implemented and writable.*

6.7. "Smdbltrp" Double Trap Extension, Version 1.0

The **Smdbltrp** extension addresses a double trap (See [Section 3.1.6.2](#)) in M-mode. When the **Smrnmi** extension ([Section 6.5](#)) is implemented, it enables invocation of the RNMI handler on a double trap in M-mode to handle the critical error. If the **Smrnmi** extension is not implemented or if a double trap occurs during the RNMI handler's execution, this extension helps transition the hart to a critical error state and enables signaling the critical error to the platform.

To improve error diagnosis and resolution, this extension supports debugging harts in a critical error state. The extension introduces a mechanism to enter Debug Mode instead of asserting a critical-error signal to the platform when the hart is in a critical error state. See ([The RISC-V Debug Specification, n.d.](#)) for details.

See [Section 3.1.6.2](#) for the operational details.

6.8. "Smctr" Control Transfer Records Extension, Version 1.0

A method for recording control flow transfer history is valuable not only for performance profiling but also for debugging. Control flow transfers refer to jump instructions (including function calls and returns), taken branch instructions, traps, and trap returns. Profiling tools, such as Linux perf, collect control transfer history when sampling software execution, thereby enabling tools, like AutoFDO, to identify hot paths for optimization.

Control flow trace capabilities offer very deep transfer history, but the volume of data produced can result in significant performance overheads due to memory bandwidth consumption, buffer management, and decoder overhead. The Control Transfer Records (CTR) extension provides a method to record a limited history in register-accessible internal chip storage, with the intent of dramatically reducing the performance overhead and complexity of collecting transfer history.

CTR defines a circular (FIFO) buffer. Each buffer entry holds a record for a single recorded control flow transfer. The number of records that can be held in the buffer depends upon both the implementation (the maximum supported depth) and the CTR configuration (the software selected depth).

Only qualified transfers are recorded. Qualified transfers are those that meet the filtering criteria, which include the privilege mode and the transfer type.

Recorded transfers are inserted at the write pointer, which is then incremented, while older recorded transfers may be overwritten once the buffer is full. Or the user can enable RAS (Return Address Stack) emulation mode, where only function calls are recorded, and function returns pop the last call record. The

source PC, target PC, and some optional metadata (transfer type, elapsed cycles) are stored for each recorded transfer.

The CTR buffer is accessible through an indirect CSR interface, such that software can specify which logical entry in the buffer it wishes to read or write. Logical entry 0 always corresponds to the youngest recorded transfer, followed by entry 1 as the next youngest, and so on.

The machine-level extension, **Smctr**, encompasses all newly added Control Status Registers (CSRs), instructions, and behavior modifications for a hart across all privilege levels. The corresponding supervisor-level extension, **Ssctr**, is essentially identical to **Smctr**, except that it excludes machine-level CSRs and behaviors not intended to be directly accessible at the supervisor level.

Smctr and **Ssctr** depend on both the implementation of S-mode and the **Ssctrind** extension.

6.8.1. CSRs

6.8.1.1. Machine Control Transfer Records Control Register (**mctrctl**)

The **mctrctl** register is a 64-bit read/write register that enables and configures the CTR capability.

63		60				59				56					
Custom						WPRI									
55		WPRI										48			
47		46		45		44		43		42		41		40	
DIRLJMPINH	INDLJMPINH	RETINH		CORSWAPINH		DIRJMPINH		INDJMPINH		DIRCALLINH		INDCALLINH			
39		38		37		36		35		34		33		32	
WPRI				TKBRINH		NTBREN		TRETINH		INTRINH		EXCINH		WPRI	
31		WPRI										24			
23		WPRI										16			
15		13				12		11		10		9		8	
WPRI				LCOFIFRZ		BPFRZ		WPRI		MTE		STE			
7		6		3				2		1		0			
RASEMU		WPRI						M		S		U			

Figure 123. Machine Control Transfer Records Control Register Format

Table 142. Machine Control Transfer Records Control Register Field Definitions

Field	Description
M, S, U	Enable transfer recording in the selected privileged mode(s).
RASEMU	Enables RAS (Return Address Stack) Emulation Mode. See Section 6.8.5.4 .
MTE	Enables recording of traps to M-mode when M=0. See Section 6.8.5.1.2 .
STE	Enables recording of traps to S-mode when S=0. See Section 6.8.5.1.2 .
BPFRZ	Set sctrstatus.FROZEN on a breakpoint exception that traps to M-mode or S-mode. See Section 6.8.5.5 .
LCOFIFRZ	Set sctrstatus.FROZEN on local-counter-overflow interrupt (LCOFI) that traps to M-mode or S-mode. See Section 6.8.5.5 .
EXCINH	Inhibit recording of exceptions. See Section 6.8.5.2 .
INTRINH	Inhibit recording of interrupts. See Section 6.8.5.2 .
TRETINH	Inhibit recording of trap returns. See Section 6.8.5.2 .
NTBREN	Enable recording of not-taken branches. See Section 6.8.5.2 .
TKBRINH	Inhibit recording of taken branches. See Section 6.8.5.2 .
INDCALLINH	Inhibit recording of indirect calls. See Section 6.8.5.2 .
DIRCALLINH	Inhibit recording of direct calls. See Section 6.8.5.2 .
INDJMPINH	Inhibit recording of indirect jumps (without linkage). See Section 6.8.5.2 .
DIRJMPINH	Inhibit recording of direct jumps (without linkage). See Section 6.8.5.2 .
CORSWAPINH	Inhibit recording of co-routine swaps. See Section 6.8.5.2 .
RETINH	Inhibit recording of function returns. See Section 6.8.5.2 .
INDLJMPINH	Inhibit recording of other indirect jumps (with linkage). See Section 6.8.5.2 .
DIRLJMPINH	Inhibit recording of other direct jumps (with linkage). See Section 6.8.5.2 .
Custom[3:0]	WARL bits designated for custom use. The value 0 must correspond to standard behavior. See Section 6.8.6 .

All fields are optional except for M, S, U, and BPFRZ. All unimplemented fields are read-only 0, while all implemented fields are writable. If the Sscfpmf extension is implemented, LCOFIFRZ must be writable.



*Because the ROI of CTR is perceived to be low for RV32 implementations, CTR does not fully support RV32. While control flow transfers in RV32 can be recorded, RV32 cannot access **xctrctl** bits 63:32. A future extension could add support for RV32 by adding 3 new CSRs (**mctrctlh**, **sctrctlh**, and **vsctrctlh**) to provide this access.*

6.8.1.2. Supervisor Control Transfer Records Control Register (**sctrctl**)

The **sctrctl** register provides supervisor mode access to a subset of **mctrctl**.

Bits 2 and 9 in **sctrctl** are read-only 0. As a result, the M and MTE fields in **mctrctl** are not accessible through **sctrctl**. All other **mctrctl** fields are accessible through **sctrctl**.

6.8.1.3. Virtual Supervisor Control Transfer Records Control Register (**vsctrctl**)

If the H extension is implemented, the **vsctrctl** register is a 64-bit read/write register that is VS-mode's version of supervisor register **sctrctl**. When V=1, **vsctrctl** substitutes for the usual **sctrctl**, so instructions that normally read or modify **sctrctl** actually access **vsctrctl** instead.

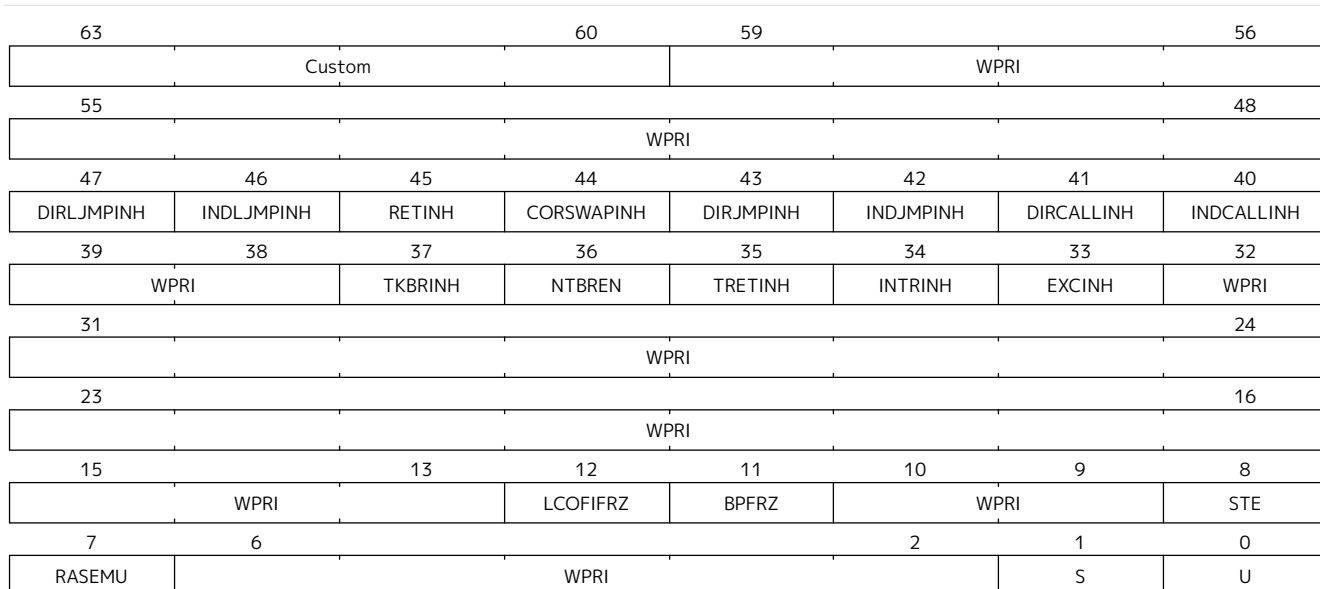


Figure 124. Virtual Supervisor Control Transfer Records Control Register Format

Table 143. Virtual Supervisor Control Transfer Records Control Register Field Definitions

Field	Description
S	Enable transfer recording in VS-mode.
U	Enable transfer recording in VU-mode.
STE	Enables recording of traps to VS-mode when S=0. See Section 6.8.5.1.2 .
BPFRZ	Set sctrstatus.FROZEN on a breakpoint exception that traps to VS-mode. See Section 6.8.5.5 .
LCOFIFRZ	Set sctrstatus.FROZEN on local-counter-overflow interrupt (LCOFI) that traps to VS-mode. See Section 6.8.5.5 .
Other field definitions match those of sctrctl . The optional fields implemented in vsctrctl should match those implemented in sctrctl .	



Unlike the CTR status register or the CTR entry registers, the CTR control register has a VS-mode version. This allows a guest to manage the CTR configuration directly, without requiring traps to HS-mode, while ensuring that the guest configuration (most notably the privilege mode enable bits) do not impact CTR behavior when V=0.

6.8.1.4. Supervisor Control Transfer Records Depth Register (**sctrdepth**)

The 32-bit **sctrdepth** register specifies the depth of the CTR buffer.

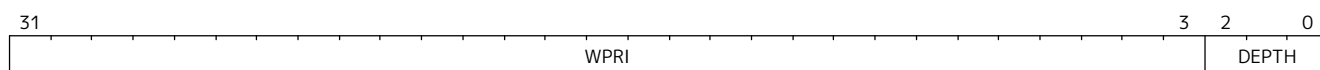


Figure 125. Supervisor Control Transfer Records Depth Register Format

Table 144. Supervisor Control Transfer Records Depth Register Field Definitions

Field	Description
DEPTH	WARL field that selects the depth of the CTR buffer. Encodings: '000 - 16 '001 - 32 '010 - 64 '011 - 128 '100 - 256 '11x - reserved The depth of the CTR buffer dictates the number of entries to which the hardware records transfers. For a depth of N, the hardware records transfers to entries 0..N-1. All Entry Registers read as 'O' and are read-only when the selected entry is in the range N to 255. When the depth is increased, the newly accessible entries contain unspecified but legal values. It is implementation-specific which DEPTH value(s) are supported.

Attempts to access **sctrdepth** from VS-mode or VU-mode raise a virtual-instruction exception, unless CTR state enable access restrictions apply. See [Section 6.8.4](#).



*It is expected that operating systems (OSs) will access **sctrdepth** only at boot, to select the maximum supported depth value. More frequent accesses may result in reduced performance in virtualization scenarios, as a result of traps from VS-mode incurred.*

There may be scenarios where software chooses to operate on only a subset of the entries, to reduce overhead. In such cases tools may choose to read only the lower entries, and OSs may choose to save/restore only on the lower entries while using SCTRCLR to clear the others.

*The value in configurable depth lies in supporting VM migration. It is expected that a platform spec may specify that one or more CTR depth values must be supported. A hypervisor may wish to restrict guests to using one of these required depths, in order to ensure that such guests can be migrated to any system that complies with the platform spec. The trapping behavior specified for VS-mode accesses to **sctrdepth** ensures that the hypervisor can impose such restrictions.*

6.8.1.5. Supervisor Control Transfer Records Status Register (**sctrstatus**)

The 32-bit **sctrstatus** register grants access to CTR status information and is updated by the hardware whenever CTR is active. CTR is active when the current privilege mode is enabled for recording and CTR is not frozen.

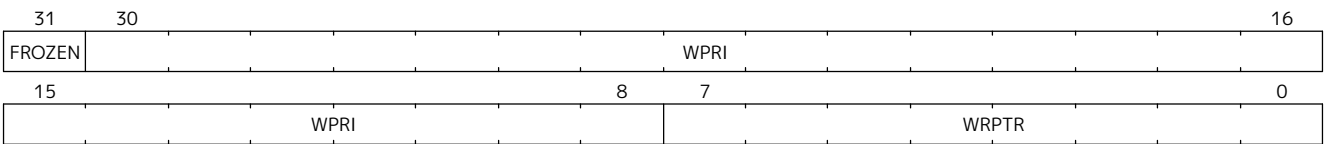


Figure 126. Supervisor Control Transfer Records Status Register Format

Table 145. Supervisor Control Transfer Records Status Register Field Definitions

Field	Description
WRPTR	WARL field that indicates the physical CTR buffer entry to be written next. It is incremented after new transfers are recorded (see Section 6.8.5), though there are exceptions when $\mathbf{xctrctl.RASEMU}=1$, see Section 6.8.5.4. For a given CTR depth (where $\text{depth} = 2^{(\text{DEPTH}+4)}$), WRPTR wraps to 0 on an increment when the value matches $\text{depth}-1$, and to $\text{depth}-1$ on a decrement when the value is 0. Bits above those needed to represent $\text{depth}-1$ (e.g., bits 7:4 for a depth of 16) are read-only 0. On depth changes, WRPTR holds an unspecified but legal value.
FROZEN	Inhibit transfer recording. See Section 6.8.5.5.

Undefined bits in **sctrstatus** are WPRI. Status fields may be added by future extensions, and software should ignore but preserve any fields that it does not recognize. Undefined bits must be implemented as read-only 0, unless a custom extension is implemented and enabled (see Section 6.8.6).



Logical entry 0, accessed via **sireg*** when **siselect**=0x200, is always the physical buffer entry preceding the WRPTR entry. More generally, the physical buffer entry *Y* associated with logical entry *X* ($X < \text{depth}$) can be determined using the formula $Y = (\text{WRPTR} - X - 1) \% \text{depth}$, where $\text{depth} = 2^{(\text{DEPTH}+4)}$. Logical entries $\geq \text{depth}$ are read-only 0.



Because the **sctrstatus** register is updated by hardware, writes should be performed with caution. If a multi-instruction read-modify-write to **sctrstatus** is performed while CTR is active, and between the read and write a qualified transfer or trap that causes CTR freeze completes, a hardware update could be lost. Software may wish to ensure that CTR is inactive before performing a read-modify-write, by ensuring that either **sctrstatus.FROZEN**=1, or that the current privilege mode is not enabled for recording.

When restoring CTR state, **sctrstatus** should be written before CTR entry state is restored. This ensures that the software writes to logical CTR entries modify the proper physical entries.



Exposing the WRPTR provides a more efficient means for synthesizing CTR entries. If a qualified control transfer is emulated, the emulator can simply increment the WRPTR, then write the synthesized record to logical entry 0. If a qualified function return is emulated while **RASEMU**=1, the emulator can clear **ctrsource.V** for logical entry 0, then decrement the WRPTR.

Exposing the WRPTR may also allow support for Linux perf's [stack stitching](#) capability.



Smctr/**Ssctr** depends upon implementation of S-mode because much of CTR state is accessible only through S-mode CSRs. If, in the future, it becomes desirable to remove this dependency, an extension could add **mctrdepth** and **mctrstatus** CSRs that reflect the same state as **sctrdepth** and **sctrstatus**, respectively. Further, such an extension should make CTR entries accessible via **miselect**/**mireg***. See Section 6.8.2.

6.8.2. Entry Registers

Control transfer records are stored in a CTR buffer, such that each buffer entry stores information about a single transfer. The CTR buffer entries are logically accessed via the indirect register access mechanism defined by the Sscsrind extension. The **siselect** index range 0x200 through 0x2FF is reserved for CTR logical entries 0 through 255. When **siselect** holds a value in this range, **sireg** provides access to **ctrsource**, **sireg2** provides access to **ctrtarget**, and **sireg3** provides access to **ctrdata**. **sireg4**, **sireg5**, and **sireg6** are read-only 0.

When **vsiselect** holds a value in 0x200..0x2FF, the **vsireg*** registers provide access to the same CTR entry register state as the analogous **sireg*** registers. There is not a separate set of entry registers for V=1.

See [Section 6.8.4](#) for cases where CTR accesses from S-mode and VS-mode may be restricted.

6.8.2.1. Control Transfer Record Source Register (**ctrsource**)

The **ctrsource** register contains the source program counter, which is the **pc** of the recorded control transfer instruction, or the **epc** of the recorded trap. The valid (**V**) bit is set by the hardware when a transfer is recorded in the selected CTR buffer entry, and implies that data in **ctrsource**, **ctrtarget**, and **ctrdata** is valid for this entry.

ctrsource is an MXLEN-bit WARL register that must be able to hold all valid virtual or physical addresses that can serve as a **pc**. It need not be able to hold any invalid addresses; implementations may convert an invalid address into a valid address that the register is capable of holding. When $XLEN < MXLEN$, both explicit writes (by software) and implicit writes (for recorded transfers) will be zero-extended.

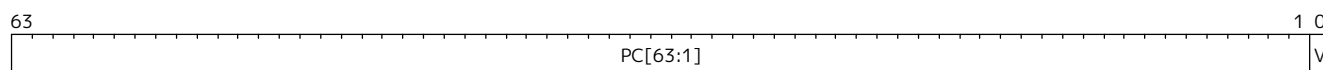


Figure 127. Control Transfer Record Source Register Format for MXLEN=64



*CTR entry registers are defined as MXLEN, despite the **xireg*** CSRs used to access them being XLEN, to ensure that entries recorded in RV64 are not truncated, as a result of CSR Width Modulation, on a transition to RV32.*

6.8.2.2. Control Transfer Record Target Register (**ctrtarget**)

The **ctrtarget** register contains the target (destination) program counter of the recorded transfer. For a not-taken branch, **ctrtarget** holds the PC of the next sequential instruction following the branch. The optional MISP bit is set by the hardware when the recorded transfer is an instruction whose target or taken/not-taken direction was mispredicted by the branch predictor. MISP is read-only 0 when not implemented.

ctrtarget is an MXLEN-bit WARL register that must be able to hold all valid virtual or physical addresses that can serve as a **pc**. It need not be able to hold any invalid addresses; implementations may convert an invalid address into a valid address that the register is capable of holding. When $XLEN < MXLEN$, both explicit writes (by software) and implicit writes (by recorded transfers) will be zero-extended.

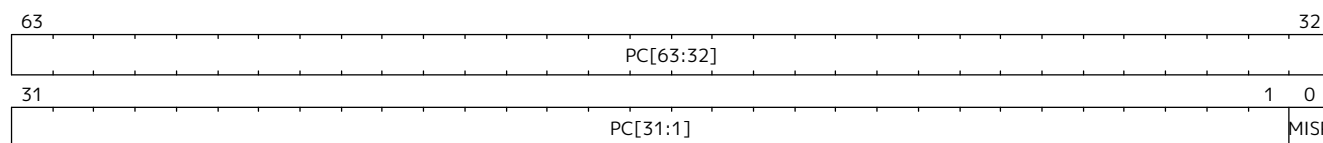


Figure 128. Control Transfer Record Target Register Format for MXLEN=64

6.8.2.3. Control Transfer Record Metadata Register (**ctrdata**)

The **ctrdata** register contains metadata for the recorded transfer. This register must be implemented, though all fields within it are optional. Unimplemented fields are read-only 0. **ctrdata** is a 64-bit register.

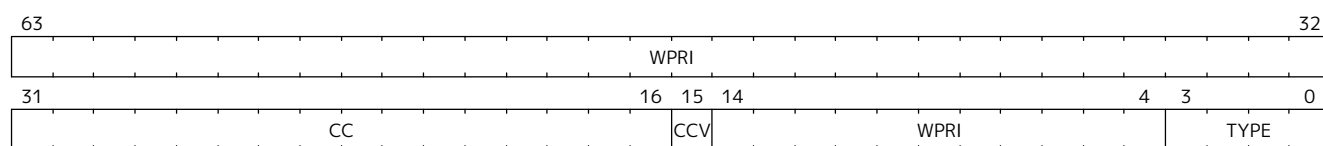


Figure 129. Control Transfer Record Metadata Register Format

Table 146. Control Transfer Record Metadata Register Field Definitions

Field	Description	Access
TYPE[3:0]	Identifies the type of the control flow transfer recorded in the entry, using the encodings listed in Table 149. Implementations that do not support this field will report 0.	WARL
CCV	Cycle Count Valid. See Section 6.8.5.3.	WARL
CC[15:0]	Cycle Count, composed of the Cycle Count Exponent (CCE, in CC[15:12]) and Cycle Count Mantissa (CCM, in CC[11:0]). See Section 6.8.5.3.	WARL

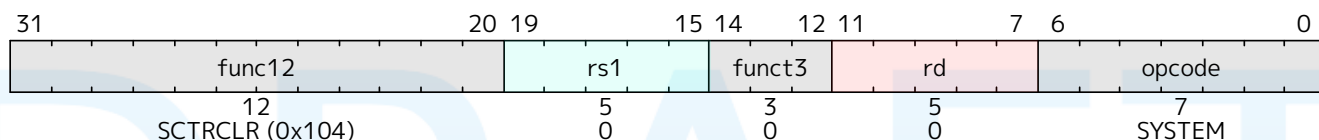
Undefined bits in `ctrdata` are WPRI. Undefined bits must be implemented as read-only 0, unless a [custom extension](#) is implemented and enabled.



Like the [Transfer Type Filtering](#) bits in `mctrctl`, the `ctrdata.TYPE` bits leverage the E-trace itype encodings.

6.8.3. Instructions

6.8.3.1. Supervisor CTR Clear Instruction



The SCTRCLR instruction performs the following operations:

- Zeroes all CTR [Entry Registers](#), for all DEPTH values
- Zeroes the CTR cycle counter and CCV (see [Section 6.8.5.3](#))

Any read of `ctrsource`, `ctrtarget`, or `ctrdata` that follows SCTRCLR, such that it precedes the next qualified control transfer, will return the value 0. Further, the first recorded transfer following SCTRCLR will have `ctrdata.CCV=0`.

SCTRCLR raises an illegal-instruction exception in U-mode, and a virtual-instruction exception in VU-mode, unless CTR state enable access restrictions apply. See [Section 6.8.4](#).

6.8.4. State Enable Access Control

When `Smstateen` is implemented, the `mstateen0.CTR` bit controls access to CTR register state from privilege modes less privileged than M-mode. When `mstateen0.CTR=1`, accesses to CTR register state behave as described in [Section 6.4.2](#) and [Section 6.8.2](#) above, while SCTRCLR behaves as described in [Section 6.8.3.1](#). When `mstateen0.CTR=0` and the privilege mode is less privileged than M-mode, the following operations raise an illegal-instruction exception:

- Attempts to access `sctrctl`, `vsctrctl`, `sctrdepth`, or `sctrstatus`
- Attempts to access `sireg*` when `siselect` is in 0x200..0x2FF, or `vsireg*` when `vsiselect` is in 0x200..0x2FF
- Execution of the SCTRCLR instruction

When `mstateen0.CTR=0`, qualified control transfers executed in privilege modes less privileged than M-mode will continue to implicitly update entry registers and `sctrstatus`.

If the H extension is implemented and `mstateen0.CTR=1`, the `hstateen0.CTR` bit controls access to supervisor CTR state when `V=1`. This state includes `sctrctl` (really `vsctrctl`), `sctrstatus`, and `sireg*` (really `vsireg*`) when `siselect` (really `vsiselect`) is in `0x200..0x2FF`. `hstateen0.CTR` is read-only 0 when `mstateen0.CTR=0`.

When `mstateen0.CTR=1` and `hstateen0.CTR=1`, VS-mode accesses to supervisor CTR state behave as described in [Section 6.4.2](#) and [Section 6.8.2](#) above, while `SCTRCLR` behaves as described in [Section 6.8.3.1](#). When `mstateen0.CTR=1` and `hstateen0.CTR=0`, both VS-mode accesses to supervisor CTR state and VS-mode execution of `SCTRCLR` raise a virtual-instruction exception.



`sctrdepth` is not included in the above list of supervisor CTR state controlled by `hstateen0.CTR` since accesses to `sctrdepth` from VS-mode raise a virtual-instruction exception regardless of the value of `hstateen0.CTR`.

When `hstateen0.CTR=0`, qualified control transfers executed while `V=1` will continue to implicitly update entry registers and `sctrstatus`.



See [Section 6.2](#) for how bit 60 in `mstateen0` and `hstateen0` can also restrict access to `sireg*/siselect` and `vsireg*/vsiselect` from privilege modes less privileged than M-mode.



Implementations that support `Smctr/Ssctr` but not `Smstateen/Ssstateen` may observe reduced performance. Because `Smctr/Ssctr` introduces a significant number of new CSRs, it is desirable to avoid save/restore of CTR state when possible. A hypervisor is likely to leverage State Enable to trap on the initial guest access to CTR state, delegating CTR and enabling save/restore of guest CTR state only once the guest has begun to use it. Without `Smstateen/Ssstateen`, a hypervisor is required to save/restore guest CTR state on every context switch.

6.8.5. Behavior

CTR records qualified control transfers. Control transfers are qualified if they meet the following criteria:

- The current privilege mode is enabled
- The transfer type is not inhibited
- `sctrstatus.FROZEN` is not set
- The transfer completes/retires

Such qualified transfers update the [Entry Registers](#) at logical entry 0. As a result, older entries are pushed down the stack; the record previously in logical entry 0 moves to logical entry 1, the record in logical entry 1 moves to logical entry 2, and so on. If the CTR buffer is full, the oldest recorded entry (previously at entry depth-1) is lost.

Recorded transfers will set the `ctrsource.V` bit to 1, and will update all implemented record fields.



In order to collect accurate and representative performance profiles while using CTR, it is recommended that hardware recording of control transfers incurs no added performance overhead, e.g., in the form of retirement or instruction execution restrictions that are not present when CTR is not active.

6.8.5.1. Privilege Mode Transitions

Transfers that change the privilege mode are a special case. What is recorded, if anything, depends on

whether the source privilege mode and/or target privilege mode are enabled for recording, and on the transfer type (trap or trap return).

Traps between enabled privilege modes are recorded as normal. Traps from a disabled privilege mode to an enabled privilege mode are partially recorded, such that the `ctrsource.PC` is 0. Traps from an enabled mode to a disabled mode, known as external traps, are not recorded by default. See [Section 6.8.5.1.2](#) for how they can be recorded.

Trap returns have similar treatment. Trap returns between enabled privilege modes are recorded as normal. Trap returns from an enabled mode back to a disabled mode are partially recorded, such that `ctrtarget.PC` is 0. Trap returns from a disabled mode to an enabled mode are not recorded.



If privileged software is configuring CTR on behalf of less privileged software, it should ensure that its privilege mode enable bit (e.g., `sctrctl.S` for Supervisor software) is cleared before a trap return to the less privileged mode. Otherwise the trap return will be recorded, leaking the privileged source `pc`.

Recording in Debug Mode is always inhibited. Transfers into and out of Debug Mode are never recorded.

The table below provides details on recording of privilege mode transitions. Standard dependencies on FROZEN and transfer type inhibits also apply, but are not covered by the table.

Table 147. Trap and Trap Return Recording

Transfer Type	Source Mode	Target Mode	
		Enabled	Disabled
Trap	Enabled	Recorded.	External trap. Not recorded by default, but see Section 6.8.5.1.2 .
	Disabled	Recorded, <code>ctrsource.PC</code> is 0.	Not recorded.
Trap Return	Enabled	Recorded.	Recorded, <code>ctrtarget.PC</code> is 0.
	Disabled	Not recorded.	Not recorded.

Virtualization Mode Transitions

Transitions between VS/VU-mode and M/HS-mode are unique in that they effect a change in the active CTR control register, and hence the CTR configuration. What is recorded, if anything, on these virtualization mode transitions depends upon fields from both `[ms]ctrctl` and `vsctrctl`.

- `mctrctl.M`, `sctrctl.S`, and `vsctrctl.{S,U}` are used to determine whether the source and target modes are enabled;
- `mctrctl.MTE`, `sctrctl.STE`, and `vsctrctl.STE` are used to determine whether an external trap is recorded (see [Section 6.8.5.1.2](#));
- `sctrctl.LCOFIFRZ` and `sctrctl.BPFRZ` determine whether CTR becomes frozen (see [Section 6.8.5.5](#))
- For all other `xctrctl` fields, the value in `vsctrctl` is used.



Consider an exception that traps from VU-mode to HS-mode, with `vsctrctl.U=1` and `sctrctl.S=1`. Because both the source mode and target mode are enabled for recording, whether the trap is recorded then depends on the CTR configuration (e.g., the [transfer type filter](#) bits) in `vsctrctl`, not in `sctrctl`.

External Traps

External traps are traps from a privilege mode enabled for CTR recording to a privilege mode that is not enabled for CTR recording. By default external traps are not recorded, but privileged software running in the target mode of the trap can opt-in to allowing CTR to record external traps into that mode. The `xctrctl.xTE` bits allow M-mode, S-mode, and VS-mode to opt-in separately.

External trap recording depends not only on the target mode, but on any intervening modes, which are modes that are more privileged than the source mode but less privileged than the target mode. Not only must the external trap enable bit for the target mode be set, but the external trap enable bit(s) for any intervening modes must also be set. See the table below for details.



Requiring intervening modes to be enabled for external traps simplifies software management of CTR. Consider a scenario where S-mode software is configuring CTR for U-mode contexts A and B, such that external traps (to any mode) are enabled for A but not for B. When switching between the two contexts, S-mode can simply toggle `sctrctl.STE`, rather than requiring a trap to M-mode to additionally toggle `mctrctl.MTE`.

This method does not provide the flexibility to record external traps to a more privileged mode but not to all intervening mode(s). Because it is expected that profiling tools generally wish to observe all external traps or none, this is not considered a meaningful limitation.

Table 148. External Trap Enable Requirements

Source Mode	Target Mode	External Trap Enable(s) Required
U-mode	S-mode	<code>sctrctl.STE</code>
	M-mode	<code>mctrctl.MTE</code> , <code>sctrctl.STE</code>
S-mode	M-mode	<code>mctrctl.MTE</code>
VU-mode	VS-mode	<code>vsctrctl.STE</code>
	HS-mode	<code>sctrctl.STE</code> , <code>vsctrctl.STE</code>
	M-mode	<code>mctrctl.MTE</code> , <code>sctrctl.STE</code> , <code>vsctrctl.STE</code>
VS-mode	HS-mode	<code>sctrctl.STE</code>
	M-mode	<code>mctrctl.MTE</code> , <code>sctrctl.STE</code>

In records for external traps, the `ctrtarget.PC` is 0.



No mechanism exists for recording external trap returns, because the external trap record includes all relevant information, and gives the trap handler (e.g., an emulator) the opportunity to modify the record.



Note that external trap recording does not depend on `EXCINH`/`INTRINH`. Thus, when external traps are enabled, both external interrupts and external exceptions are recorded.

STE allows recording of traps from U-mode to S-mode as well as from VS/VU-mode to HS-mode. The hypervisor can flip `sctrctl.STE` before entering a guest if it wants different behavior for U-to-S vs VS/VU-to-HS.

If external trap recording is implemented, `mctrctl.MTE` and `sctrctl.STE` must be implemented, while `vsctrctl.STE` must be implemented if the H extension is implemented.

6.8.5.2. Transfer Type Filtering

Default CTR behavior, when all transfer type filter bits (`xctrctl[47:32]`) are unimplemented or 0, is to

record all control transfers within enabled privileged modes. By setting transfer type filter bits, software can opt out of recording select transfer types, or opt into recording non-default operations. All transfer type filter bits are optional.



Because not-taken branches are not recorded by default, the polarity of the associated enable bit (NTBREN) is the opposite of other bits associated with transfer type filtering (TKBRINH, RETINH, etc). Non-default operations require opt-in rather than opt-out.

The transfer type filter bits leverage the type definitions specified in the [RISC-V Efficient Trace Spec v2.0](#) (Table 4.4 and Section 4.1.1). For completeness, the definitions are reproduced below.



Here "indirect" is used interchangeably with "uninferrable", which is used in the trace spec. Both imply that the target of the jump is not encoded in the opcode.

Table 149. Control Transfer Type Definitions

Encoding	Transfer Type Name
0	Not used by CTR
1	Exception
2	Interrupt
3	Trap return
4	Not-taken branch
5	Taken branch
6	<i>reserved</i>
7	<i>reserved</i>
8	Indirect call
9	Direct call
10	Indirect jump (without linkage)
11	Direct jump (without linkage)
12	Co-routine swap
13	Function return
14	Other indirect jump (with linkage)
15	Other direct jump (with linkage)

Encodings 8 through 15 refer to various encodings of jump instructions. The types are distinguished as described below.

Table 150. Control Transfer Type Definitions

Transfer Type Name	Associated Opcodes
Indirect call	JALR $x1, rs$ where $rs \neq x5$
	JALR $x5, rs$ where $rs \neq x1$
	CJALR $rs1$ where $rs1 \neq x5$
Direct call	JAL $x1$
	JAL $x5$
	CJAL
	CM.JALT <i>index</i>
Indirect jump (without linkage)	JALR $x0, rs$ where $rs \neq (x1 \text{ or } x5)$
	CJR $rs1$ where $rs1 \neq (x1 \text{ or } x5)$
Direct jump (without linkage)	JAL $x0$
	CJ
	CM.JT <i>index</i>
Co-routine swap	JALR $x1, x5$
	JALR $x5, x1$
	CJALR $x5$
Function return	JALR rd, rs where $rs == (x1 \text{ or } x5)$ and $rd \neq (x1 \text{ or } x5)$
	CJR $rs1$ where $rs1 == (x1 \text{ or } x5)$
	CM.POPRET(<i>Z</i>)
Other indirect jump (with linkage)	JALR rd, rs where $rs \neq (x1 \text{ or } x5)$ and $rd \neq (x0, x1, \text{ or } x5)$
Other direct jump (with linkage)	JAL rd where $rd \neq (x0, x1, \text{ or } x5)$



If implementation of any transfer type filter bit results in reduced software performance, perhaps due to additional retirement restrictions, it is strongly recommended that this reduced performance apply only when the bit is set. Alternatively, support for the bit may be omitted. Maintaining software performance for the default CTR configuration, when all transfer type bits are cleared, is recommended.

6.8.5.3. Cycle Counting

The **ctrdata** register may optionally include a count of CPU cycles elapsed since the prior CTR record. The elapsed cycle count value is represented by the CC field, which has a 12-bit mantissa component (Cycle Count Mantissa, or CCM) and a 4-bit exponent component (Cycle Count Exponent, or CCE).

The elapsed cycle counter (CtrCycleCounter) increments at the same rate as the **mcycle** counter. Only cycles while CTR is active are counted, where active implies that the current privilege mode is enabled for recording and CTR is not frozen. The CC field is encoded such that CCE holds 0 if the CtrCycleCounter value is less than 4096, otherwise it holds the index of the most significant one bit in the CtrCycleCounter value, minus 11. CCM holds CtrCycleCounter bits CCE+10:CCE-1.

The elapsed cycle count can then be calculated by software using the following formula:

```

if (CCE==0):
    return CCM
else:
    return (212 + CCM) << CCE-1
endif

```

The CtrCycleCounter is reset on writes to **xctrctl**, and on execution of SCTRCLR, to ensure that any accumulated cycle counts do not persist across a context switch.

An implementation that supports cycle counting must implement CCV and all CCM bits, but may implement 0.4 exponent bits in CCE. Unimplemented CCE bits are read-only 0. For implementations that support transfer type filtering, it is recommended to implement at least 3 exponent bits. This allows capturing the full latency of most functions, when recording only calls and returns.

The size of the CtrCycleCounter required to support each CCE width is given in the table below.

Table 151. Cycle Counter Size Options

CCE bits	CtrCycleCounter bits	Max elapsed cycle value
0	12	4095
1	13	8191
2	15	32764
3	19	524224
4	27	134201344

When $CCE > 1$, the granularity of the reported cycle count is reduced. For example, when $CCE = 3$, the bottom 2 bits of the cycle counter are not reported, and thus the reported value increments only every 4 cycles. As a result, the reported value represents an undercount of elapsed cycles for most cases (when the unreported bits are non-zero). On average, the undercount will be $(2^{CCE-1}-1)/2$. Software can reduce the average undercount to 0 by adding $(2^{CCE-1}-1)/2$ to each computed cycle count value when $CCE > 1$.

Though this compressed method of representation results in some imprecision for larger cycle count values, it produces meaningful area savings, reducing storage per entry from 27 bits to 16.

The CC value saturates when all implemented bits in CCM and CCE are 1.

The CC value is valid only when the Cycle Count Valid (CCV) bit is set. If $CCV = 0$, the CC value might not hold the correct count of elapsed active cycles since the last recorded transfer. The next record will have $CCV = 0$ after a write to **xctrctl**, or execution of SCTRCLR, since CtrCycleCounter is reset. CCV should additionally be cleared after any other implementation-specific scenarios where active cycles might not be counted in CtrCycleCounter.

6.8.5.4. RAS (Return Address Stack) Emulation Mode

When the optional **xctrctl**.RASEMU bit is implemented and set to 1, transfer recording behavior is altered to emulate the behavior of a return-address stack (RAS).

- Indirect and direct calls are recorded as normal
- Function returns pop the most recent call, by decrementing the WRPTR then invalidating the WRPTR

entry (by setting `ctrsource.V=0`). As a result, logical entry 0 is invalidated and moves to logical entry depth-1, while logical entries 1..depth-1 move to 0..depth-2.

- Co-routine swaps affect both a return and a call. Logical entry 0 is overwritten, and `WRPTR` is not modified.
- Other transfer types are inhibited
- Transfer type filtering bits (`xctrctl[47:32]`) and external trap enable bits (`xctrctl.xTE`) are ignored

Profiling tools often collect call stacks along with each sample. Stack walking, however, is a complex and often slow process that may require recompilation (e.g., `-fno-omit-frame-pointer`) to work reliably. With RAS emulation, tools can ask CTR hardware to save call stacks even for unmodified code.



CTR RAS emulation has limitations. The CTR buffer will contain only partial stacks in cases where the call stack depth was greater than the CTR depth, CTR recording was enabled at a lower point in the call stack than `main()`, or where the CTR buffer was cleared since `main()`.

The CTR stack may be corrupted in cases where calls and returns are not symmetric, such as with stack unwinding (e.g., `setjmp/longjmp`, C++ exceptions), where stale call entries may be left on the CTR stack, or user stack switching, where calls from multiple stacks may be intermixed.



As described in [Section 6.8.5.3](#), when `CCV=1`, the `CC` field provides the elapsed cycles since the prior CTR entry was recorded. This introduces implementation challenges when `RASEMU=1` because, for each recorded call, there may have been several recorded calls (and returns which “popped” them) since the prior remaining call entry was recorded (see [Section 6.8.5.4](#)). The implication is that returns that pop a call entry not only do not reset the cycle counter, but instead add the `CC` field from the popped entry to the counter. For simplicity, an implementation may opt to record `CCV=0` for all calls, or those whose parent call was popped, when `RASEMU=1`.

6.8.5.5. Freeze

When `sctrstatus.FROZEN=1`, transfer recording is inhibited. This bit can be set by hardware, as described below, or by software.

When `sctrctl.LCOFIFRZ=1` and a local-counter-overflow interrupt (LCOFI) traps (as a result of an HPM counter overflow) to M-mode or to S-mode, `sctrstatus.FROZEN` is set by hardware. This inhibits CTR recording until software clears `FROZEN`. The LCOFI trap itself is not recorded.

Freeze on LCOFI ensures that the execution path leading to the sampled instruction (`xepc`) is preserved, and that the local-counter-overflow interrupt (LCOFI) and associated Interrupt Service Routine (ISR) do not displace any recorded transfer history state. It is the responsibility of the ISR to clear `FROZEN` before `xRET`, if continued control transfer recording is desired.



LCOFI refers only to architectural traps directly caused by a local counter overflow. If a local-counter-overflow interrupt is recognized without a trap, `FROZEN` is not automatically set. For instance, no freeze occurs if the LCOFI is pended while interrupts are masked, and software recognizes the LCOFI (perhaps by reading `stopi` or `sip`) and clears `sip.LCOFIP` before the trap is raised. As a result, some or all CTR history may be overwritten while handling the LCOFI. Such cases are expected to be very rare; for most usages (e.g., application profiling) privilege mode filtering is sufficient to ensure that CTR updates are inhibited while interrupts are handled in a more privileged mode.

Similarly, on a breakpoint exception that traps to M-mode or S-mode with `sctrctl.BPFRZ=1`, FROZEN is set by hardware. The breakpoint exception itself is not recorded.



Breakpoint exception refers to synchronous exceptions with a cause value of Breakpoint (3), regardless of source (ebreak, c.ebreak, Sdtrig); it does not include entry into Debug Mode, even in cores where this is implemented as an exception.

If the H extension is implemented, freeze behavior for LCOFIs and breakpoint exceptions that trap to VS-mode is determined by the LCOFIFRZ and BPFRZ values, respectively, in `vsctrctl`. This includes virtual LCOFIs pended by a hypervisor.



When a guest uses the SBI Supervisor Software Events (SSE) extension, the LCOFI will trap to HS-mode, which will then invoke a registered VS-mode LCOFI handler routine. If `vsctrctl.LCOFIFRZ=1`, the HS-mode handler will need to emulate the freeze by setting `sctrstatus.FROZEN=1` before invoking the registered handler routine.

6.8.6. Custom Extensions

Any custom CTR extension must be associated with a non-zero value within the designated custom bits in `xctrctl`. When the custom bits hold a non-zero value that enables a custom extension, the extension may alter standard CTR behavior, and may define new custom status fields within `sctrstatus` or the CTR [Entry Registers](#). All custom status fields, and standard status fields whose behavior is altered by the custom extension, must revert to standard behavior when the custom bits hold zero. This includes read-only 0 behavior for any bits undefined by any implemented standard extensions.

6.9. 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. The Zicfiss and Zicfilp extensions provide backward-edge and forward-edge control flow integrity respectively. Please see the Control-flow Integrity chapter of the Unprivileged ISA specification for further details on these CFI capabilities and the associated Unprivileged ISA.

6.9.1. Landing Pad (Zicfilp)

This section specifies the Privileged ISA for the Zicfilp extension.

6.9.1.1. Landing-Pad-Enabled (LPE) State

The term `xLPE` is used to determine if forward-edge CFI using landing pads provided by the Zicfilp extension is enabled at a privilege mode.

When S-mode is implemented, it is determined as follows:

Table 152. `xLPE` determination when S-mode is implemented

Privilege Mode	<code>xLPE</code>
M	<code>mseccfg.MLPE</code>
S or HS	<code>menvcfg.LPE</code>
VS	<code>henvcfg.LPE</code>
U or VU	<code>senvcfg.LPE</code>

When S-mode is not implemented, it is determined as follows:

Table 153. xLPE determination when S-mode is not implemented

Privilege Mode	xLPE
M	mseccfg.MLPE
U	menvcfg.LPE



The Zicfilp must be explicitly enabled for use at each privilege mode.

Programs compiled with the LPAD instruction continue to function correctly, but without forward-edge CFI protection, when the Zicfilp extension is not implemented or is not enabled.

DRAFT

6.9.1.2. Preserving Expected Landing Pad State on Traps

A trap may need to be delivered to the same or to a higher privilege mode upon completion of **JALR/C.JR**, but before the instruction at the target of indirect call/jump was decoded, due to:

- Asynchronous interrupts.
- Synchronous exceptions with priority higher than that of a software-check exception with **xtval** set to "landing pad fault (code=2)" (See [Table 113](#) of Privileged Specification).

The software-check exception caused by **Zicfilp** has higher priority than an illegal-instruction exception but lower priority than instruction access-fault.

The software-check exception due to the instruction not being an **LPAD** instruction when **ELP** is **LP_EXPECTED** or a software-check exception caused by the **LPAD** instruction itself leads to a trap being delivered to the same or to a higher privilege mode.

In such cases, the **ELP** prior to the trap, the previous **ELP**, must be preserved by the trap delivery such that it can be restored on a return from the trap. To store the previous **ELP** state on trap delivery to M-mode, an **MPELP** bit is provided in the **mstatus** CSR. To store the previous **ELP** state on trap delivery to S/HS-mode, an **SPELP** bit is provided in the **mstatus** CSR. The **SPELP** bit in **mstatus** can be accessed through the **sstatus** CSR. To store the previous **ELP** state on traps to VS-mode, a **SPELP** bit is defined in the **vsstatus** (VS-modes version of **sstatus**). To store the previous **ELP** state on transition to Debug Mode, a **peIp** bit is defined in the **dcsr** register.

When a trap is taken into privilege mode **x**, the **xPELP** is set to **ELP** and **ELP** is set to **NO_LP_EXPECTED**.

An **MRET** or **SRET** instruction is used to return from a trap in M-mode or S-mode, respectively. When executing an **xRET** instruction, if the new privilege mode is **y**, then **ELP** is set to the value of **xPELP** if **yLPE** (see [Section 6.9.1.1](#)) is 1; otherwise, it is set to **NO_LP_EXPECTED**; **xPELP** is set to **NO_LP_EXPECTED**.

Upon entry into Debug Mode, the **peIp** bit in **dcsr** is updated with the **ELP** at the privilege level the hart was previously in, and the **ELP** is set to **NO_LP_EXPECTED**. When a hart resumes from Debug Mode, if the new privilege mode is **y**, then **ELP** is set to the value of **peIp** if **yLPE** (see [Section 6.9.1.1](#)) is 1; otherwise, it is set to **NO_LP_EXPECTED**.

See also [Section 6.5](#) for semantics added to the **RNMI** trap and the **MNRET** instruction when this extension is implemented.



*The trap handler in privilege mode **x** must save the **xPELP** bit and the **x7** register before performing an indirect call/jump if **xLPE=1**. If the privilege mode **x** can respond to interrupts and **xLPE=1**, then the trap handler should also save these values before enabling interrupts.*

*The trap handler in privilege mode **x** must restore the saved **xPELP** bit and the **x7** register before executing the **xRET** instruction to return from a trap.*

6.9.2. Shadow Stack (Zicfiss)

This section specifies the Privileged ISA for the Zicfiss extension.

6.9.2.1. Shadow Stack Pointer (ssp) CSR access control

Attempts to access the **ssp** CSR may result in either an illegal-instruction exception or a virtual-instruction exception, contingent upon the state of the **xenvcfg.SSE** fields. The conditions are specified as follows:

- If the privilege mode is less than M and **menvcfg.SSE** is 0, an illegal-instruction exception is raised.
- Otherwise, if in U-mode and **senvcfg.SSE** is 0, an illegal-instruction exception is raised.
- Otherwise, if in VS-mode and **henvcfg.SSE** is 0, a virtual-instruction exception is raised.
- Otherwise, if in VU-mode and either **henvcfg.SSE** or **senvcfg.SSE** is 0, a virtual-instruction exception is raised.
- Otherwise, the access is allowed.

6.9.2.2. Shadow-Stack-Enabled (SSE) State

The term **xSSE** is used to determine if backward-edge CFI using shadow stacks provided by the Zicfiss extension is enabled at a privilege mode.

When S-mode is implemented, it is determined as follows:

Table 154. **xSSE** determination when S-mode is implemented

Privilege Mode	xSSE
M	0
S or HS	menvcfg.SSE
VS	henvcfg.SSE
U or VU	senvcfg.SSE

When S-mode is not implemented, then **xSSE** is 0 at both M and U privilege modes.



Activating Zicfiss in U-mode must be done explicitly per process. Not activating Zicfiss at U-mode for a process when that application is not compiled with Zicfiss allows it to invoke shared libraries that may contain Zicfiss instructions. The Zicfiss instructions in the shared library revert to their Zimop/Zcmop-defined behavior in this case.

When Zicfiss is enabled in S-mode it is benign to use an operating system that is not compiled with Zicfiss instructions. Such an operating system that does not use backward-edge CFI for S-mode execution may still activate Zicfiss for U-mode applications.

When programs that use Zicfiss instructions are installed on a processor that supports the Zicfiss extension but the extension is not enabled at the privilege mode where the program executes, the program continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.

When programs that use Zicfiss instructions are installed on a processor that does not support the Zicfiss extension but supports the Zimop and Zcmop extensions, the programs continues to function correctly but without backward-edge CFI protection as the Zicfiss instructions will revert to their Zimop/Zcmop-defined behavior.

On processors that do not support Zimop/Zcmop extensions, all Zimop/Zcmop code points including those used for Zicfiss instructions may cause an illegal-instruction exception. Execution of programs that use these instructions on such machines is not supported.

Activating Zicfiss in M-mode is currently not supported. Additionally, when S-mode is not implemented, activation in U-mode is also not supported. These functionalities may be introduced in a future standard extension.



Changes to xSSE take effect immediately; address-translation caches need not be synchronized with SFENCE.VMA, HFENCE.GVMA, or HFENCE.VVMA instructions.

6.9.2.3. Shadow Stack Memory Protection

To protect shadow stack memory, the memory is associated with a new page type – the Shadow Stack (SS) page – in the single-stage and VS-stage page tables. The encoding **R=0**, **W=1**, and **X=0**, is defined to represent an SS page. When **menvcfg.SSE=0**, this encoding remains reserved. Similarly, when **V=1** and **henvcfg.SSE=0**, this encoding remains reserved at VS and VU levels.

If **satp.MODE** (or **vsatp.MODE** when **V=1**) is set to **Bare** and the effective privilege mode is less than M, shadow stack instructions raise a store/AMO access-fault exception. When the effective privilege mode is M, memory access by an **SSAMOSWAP.W/D** instruction results in a store/AMO access-fault exception.

Memory mapped as an SS page cannot be written to by instructions other than **SSAMOSWAP.W/D**, **SSPUSH**, and **C.SSPUSH**. Attempts will raise a store/AMO access-fault exception. Access to a SS page using *cache-block operation* (**CB0.***) instructions is not permitted. Such accesses will raise a store/AMO access-fault exception. Implicit accesses, including instruction fetches to an SS page, are not permitted. Such accesses will raise an access-fault exception appropriate to the access type. However, the shadow stack is readable by all instructions that only load from memory.

Stores to shadow stack pages by instructions other than SSAMOSWAP, SSPUSH, and C.SSPUSH will trigger a store/AMO access-fault exception, not a store/AMO page-fault exception, signaling a fatal error. A store/AMO page-fault suggests that the operating system could address and rectify the fault, which is not feasible in this scenario. Hence, the page-fault handler must decode the opcode of the faulting instruction to discern whether the fault was caused by a non-shadow-stack instruction writing to an SS page (a fatal condition) or by a shadow stack instruction to a non-resident page (a recoverable condition). The performance-critical nature of operating system page fault handlers necessitates triggering an access fault instead of a page fault, allowing for a straightforward distinction between fatal conditions and recoverable faults.



Operating systems must ensure that no writable, non-shadow-stack alias virtual address mappings exist for the physical memory backing the shadow stack. Furthermore, in systems where an address-misaligned exception supersedes the access-fault exception, handlers emulating misaligned stores must be designed to cause an access-fault exception when the store is directed to a shadow stack page.

All instructions that perform load operations are allowed to read from the shadow stack. This feature facilitates debugging and performance profiling by allowing examination of the link register values backed up in the shadow stack.



As of the drafting of this specification, instruction fetches are the sole type of implicit access subjected to single- or VS-stage address translation.

If a shadow stack (SS) instruction raises an access-fault, page-fault, or guest-page-fault exception that is supposed to indicate the original instruction type (load or store/AMO), then the reported exception cause is

respectively a store/AMO access fault (code 7), a store/AMO page fault (code 15), or a store/AMO guest-page fault (code 23). For shadow stack instructions, the reported instruction type is always as though it were a store or AMO, even for instructions **SSPOPCHK** and **C.SSPOPCHK** that only read from memory and do not write to it.



When Zicfiss is implemented, the existing "store/AMO" exceptions can be thought of as "store/AMO/SS" exceptions, indicating that the trapping instruction is either a store, an AMO, or a shadow stack instruction.

Shadow stack instructions are restricted to accessing shadow stack (**pte.xwr=010b**) pages. Should a shadow stack instruction access a page that is not designated as a shadow stack page and is not marked as read-only (**pte.xwr=001**), a store/AMO access-fault exception will be invoked. Conversely, if the page being accessed by a shadow stack instruction is a read-only page, a store/AMO page-fault exception will be triggered.



*Shadow stack loads and stores will trigger a store/AMO page-fault if the accessed page is read-only, to support copy-on-write (COW) of a shadow stack page. If the page has been marked read-only for COW tracking, the page-fault handler responds by creating a copy of the page and updates the **pte.xwr** to **010b**, thereby designating each copy as a shadow stack page. Conversely, if the access targets a genuinely read-only page, the fault being reported as a store/AMO page-fault signals to the operating system that the fault is fatal and non-recoverable. Reporting the fault as a store/AMO page-fault, even for **SSPOPCHK** initiated memory access, aids in the determination of fatality; if these were reported as load page-faults, access to a truly read-only page might be mistakenly treated as a recoverable fault, leading to the faulting instruction being retried indefinitely. The PTE does not provide a read-only shadow stack encoding.*

Attempts by shadow stack instructions to access pages marked as read-write, read-write-execute, read-execute, or execute-only result in a store/AMO access-fault exception, similarly indicating a fatal condition.

Shadow stacks should be bounded at each end by guard pages to prevent accidental underflows or overflows from one shadow stack into another. Conventionally, a guard page for a stack is a page that is not accessible by the process that owns the stack.

If the virtual address in **ssp** is not **XLEN** aligned, then the **SSPUSH/ C.SSPUSH/SSPOPCHK/C.SSPOPCHK** instructions cause a store/AMO access-fault exception.



Misaligned accesses to shadow stack are not required and enforcing alignment is more secure to detect errors in the program. An access-fault exception is raised instead of address-misaligned exception in such cases to indicate fatality and that the instruction must not be emulated by a trap handler.

Correct execution of shadow stack instructions that access memory requires the the accessed memory to be idempotent. If the memory referenced by **SSPUSH/C.SSPUSH/SSPOPCHK/C.SSPOPCHK/SSAMOSWAP.W/D** instructions is not idempotent, then the instructions cause a store/AMO access-fault exception.



*The **SSPOPCHK** instruction performs a load followed by a check of the loaded data value with the link register as source. If the check against the link register faults, and the instruction is restarted by the trap handler, then the instruction will perform a load again. If the memory from which the load is performed is non-idempotent, then the second load may cause unexpected side effects. Shadow stack instructions that access the shadow stack require the memory referenced by **ssp** to be idempotent to avoid such concerns. Locating shadow stacks in non-idempotent memory, such as non-idempotent device memory, is not an expected usage, and requiring memory referenced to be idempotent does not pose a significant restriction.*

The **U** and **SUM** bit enforcement is performed normally for shadow stack instruction initiated memory accesses. The state of the **MXR** bit does not affect read access to a shadow stack page as the shadow stack page is always readable by all instructions that load from memory.

The G-stage address translation and protections remain unaffected by the Zicfiss extension. The **xwr == 010b** encoding in the G-stage PTE remains reserved. When G-stage page tables are active, the shadow stack instructions that access memory require the G-stage page table to have read-write permission for the accessed memory; else a store/AMO guest-page-fault exception is raised.



A future extension may define a shadow stack encoding in the G-stage page table to support use cases such as a hypervisor enforcing shadow stack protections for its guests.

Svpbmt and Svnaptot extensions are supported for shadow stack pages.

The PMA checks are extended to require memory referenced by shadow stack instructions to be idempotent. The PMP checks are extended to require read-write permission for memory accessed by shadow stack instructions. If the PMP does not provide read-write permissions or if the accessed memory is not idempotent then a store/AMO access-fault exception is raised.

The **SSAMOSWAP.W/D** instructions require the PMA of the accessed memory range to provide AMOSwap level support.

6.10. Pointer Masking Extensions, Version 1.0.0

6.10.1. Introduction

RISC-V Pointer Masking (PM) is a feature that, when enabled, causes the CPU to ignore the upper bits of the effective address (these terms will be defined more precisely in the Background section). This allows these bits to be used in whichever way the application chooses. The version of the extension being described here specifically targets **tag checks**: When an address is accessed, the tag stored in the masked bits can be compared against a range-based tag. This is used for dynamic safety checkers such as HWASAN (Serebryany et al., 2018). Such tools can be applied in all privilege modes (U, S, and M).

HWASAN leverages tags in the upper bits of the address to identify memory errors such as use-after-free or buffer overflow errors. By storing a **pointer tag** in the upper bits of the address and checking it against a **memory tag** stored in a side table, it can identify whether a pointer is pointing to a valid location. Doing this without hardware support introduces significant overheads since the pointer tag needs to be manually removed for every conventional memory operation. Pointer masking support reduces these overheads.

Pointer masking only adds the ability to ignore pointer tags during regular memory accesses. The tag checks themselves can be implemented in software or hardware. If implemented in software, pointer masking still provides performance benefits since non-checked accesses do not need to transform the address before every memory access. Hardware implementations are expected to provide even larger benefits due to performing tag checks out-of-band and hardening security guarantees derived from these checks. We anticipate that future extensions may build on pointer masking to support this functionality in hardware.

It is worth mentioning that while HWASAN is the primary use-case for the current pointer masking extension, a number of other hardware/software features may be implemented leveraging Pointer Masking. Some of these use cases include sandboxing, object type checks and garbage collection bits in runtime systems. Note that the current version of the spec does not explicitly address these use cases, but future extensions may build on it to do so.

While we describe the high-level concepts of pointer masking as if it was a single extension, it is, in reality, a family of extensions that implementations or profiles may choose to individually include or exclude (see [Section 6.10.2.7](#)).

6.10.2. Background

6.10.2.1. Definitions

We now define basic terms. Note that these rely on the definition of an “ignore” transformation, which is defined in [Section 6.10.2.2](#).

- **Effective address (as defined in the RISC-V Base ISA):** A load/store effective address sent to the memory subsystem (e.g., as generated during the execution of load/store instructions). This does not include addresses corresponding to implicit accesses, such as page-table walks.
- **Masked bits:** The upper PMLen bits of an address, where PMLen is a configurable parameter. We will use PMLen consistently throughout this chapter to refer to this parameter.
- **Transformed address:** An effective address after the ignore transformation has been applied.
- **Address translation mode:** The MODE of the currently active address translation scheme as defined in the RISC-V privileged specification. This could, for example, refer to Bare, Sv39, Sv48, and Sv57. In accordance with the privileged specification, non-Bare translation modes are referred to as virtual-memory schemes. For the purpose of this specification, M-mode translation is treated as equivalent to Bare.
- **Address validity:** The RISC-V privileged spec defines validity of addresses based on the address translation mode that is currently in use (e.g., Sv57, Sv48, Sv39, etc.). For a virtual address to be valid, all bits in the unused portion of the address must be the same as the Most Significant Bit (MSB) of the used portion. For example, when page-based 48-bit virtual memory (Sv48) is used, load/store effective addresses, which are 64 bits, must have bits 63–48 all set to bit 47, or else a page-fault exception will occur. For physical addresses, validity means that bits XLEN-1 to PABITS are zero, where PABITS is the number of physical address bits supported by the processor.
- **NVBITS:** The upper bits within a virtual address that have no effect on addressing memory and are only used for validity checks. These bits depend on the currently active address translation mode. For

example, in Sv48, these are bits 63-48.

- **VBITS:** The bits within a virtual address that affect which memory is addressed. These are the bits of an address which are used to index into page tables.

6.10.2.2. The “Ignore” Transformation

The ignore transformation differs depending on whether it applies to a virtual or physical address. For virtual addresses, it replaces the upper PMLen bits with the sign extension of the PMLen+1st bit.

```
transformed_effective_address =
  {{PMLen{effective_address[XLEN-PMLen-1]}}, effective_address[XLEN-PMLen-1:0]}
```

Listing 26. “Ignore” Transformation for virtual addresses, expressed in Verilog code.



If PMLen is less than or equal to NVBITS for the largest supported address translation mode on a given architecture, this is equivalent to ignoring a subset of NVBITS. This enables cheap implementations that modify validity checks in the CPU instead of performing the sign extension.

When applied to a physical address, including guest-physical addresses (i.e., all cases except when the active satp register’s MODE field != Bare), the ignore transformation replaces the upper PMLen bits with 0. This includes both the case of running in M-mode and running in other privilege modes with Bare address translation mode.

```
transformed_effective_address =
  {{PMLen{0}}, effective_address[XLEN-PMLen-1:0]}
```

Listing 27. “Ignore” Transformation for physical addresses, expressed in Verilog code.



This definition is consistent with the way that RISC-V already handles physical and virtual addresses differently. While the unused upper bits of virtual addresses are the sign-extension of the used bits (see the definition of “address validity” in [Section 6.10.2.1](#)), the equivalent bits in physical addresses are zero-extended. This is necessary due to their interactions with other mechanisms such as Physical Memory Protection (PMP).

When pointer masking is enabled, the ignore transformation will be applied to every explicit memory access (e.g., loads/stores, atomics operations, and floating point loads/stores). The transformation **does not** apply to implicit accesses such as page-table walks or instruction fetches. The set of accesses that pointer masking applies to is described in [Section 6.10.2.6](#).



Pointer masking does not change the underlying address generation logic or permission checks. Under a fixed address translation mode, it is semantically equivalent to replacing a subset of instructions (e.g., loads and stores) with an instruction sequence that applies the ignore operation to the target address of this instruction and then applies the instruction to the transformed address. References to address translation and other implementation details in the text are primarily to explain design decisions and common implementation patterns.

Note that pointer masking is purely an arithmetic operation on the address that makes no assumption about the meaning of the addresses it is applied to. Pointer masking with the same value of PMLen always has the same effect for the same type of address (virtual or physical). This ensures that code that relies on pointer masking does not need to be aware of the environment it runs in once pointer masking has been enabled, as long as the value of PMLen is known, and whether or not addresses are virtual or physical. For

example, the same application or library code can run in user mode, supervisor mode or M-mode (with different address translation modes) without modification.



A common scenario for such code is that addresses are generated by `mmap` system calls. This abstracts away the details of the underlying address translation mode from the application code. Software therefore needs to be aware of the value of `PMLLEN` to ensure that its minimally required number of tag bits is supported. [Section 6.10.2.4](#) covers how this value is derived.

6.10.2.3. Example

[Table 155](#) shows an example of the pointer masking transformation on a virtual address when PM is enabled for RV64 under Sv57 (`PMLLEN=7`).

Table 155. Example of PM address translation for RV64 under Sv57

Page-based profile	Sv57 on RV64
Effective Address	0xABFFFFFF12345678 NVBITS[1010101] VBITS[111111111111111111110001...000]
PMLLEN	7
Mask	0x01FFFFFFFFFFFFFFF NVBITS[0000000] VBITS[111111111111111111111111...111]
PMLLEN+1st bit from the top (i.e., bit <code>XLEN-PMLLEN-1</code>)	1
Transformed effective address	0xFFFFFFFF12345678 NVBITS[1111111] VBITS[1111111111111111111111110001...000]

If the address was a physical address rather than a virtual address with Sv57, the transformed address with `PMLLEN=7` would be 0x1FFFFFFFF12345678.

6.10.2.4. Determining the Value of PMLLEN

From an implementation perspective, ignoring bits is deeply connected to the maximum virtual and physical address space supported by the processor (e.g., Bare, Sv48, Sv57). In particular, applying the above transformation is cheap if it covers only bits that are not used by **any** supported address translation mode (as it is equivalent to switching off validity checks). Masking NVBITS beyond those bits is more expensive as it requires ignoring them in the TLB tag, and even more expensive if the masked bits extend into the VBITS portion of the address (as it requires performing the actual sign extension). Similarly, when running in Bare or M mode, it is common for implementations to not use a particular number of bits at the top of the physical address range and fix them to zero. Applying the ignore transformation to those bits is cheap as well, since it will result in a valid physical address with all the upper bits fixed to 0.

The current standard only supports `PMLLEN=XLEN-48` (i.e., `PMLLEN=16` in RV64) and `PMLLEN=XLEN-57` (i.e., `PMLLEN=7` in RV64). A setting has been reserved to potentially support other values of `PMLLEN` in future standards. In such future standards, different supported values of `PMLLEN` may be defined for each privilege mode (U/VU, S/HS, and M).



Future versions of the pointer masking extension may introduce the ability to freely configure the value of `PMLLEN`. The current extension does not define the behavior if `PMLLEN` was different from the values defined above. In particular, there is no guarantee that a future pointer masking extension would define the ignore operation in the same way for those values of `PMLLEN`.

6.10.2.5. Pointer Masking and Privilege Modes

Pointer masking is controlled separately for different privilege modes. The subset of supported privilege modes is determined by the set of supported pointer masking extensions. Different privilege modes may have different pointer masking settings active simultaneously and the hardware will automatically apply the pointer masking settings of the currently active privilege mode. A privilege mode's pointer masking setting is configured by bits in configuration registers of the next-higher privilege mode.

Note that the pointer masking setting that is applied only depends on the active privilege mode, not on the address that is being masked. Some operating systems (e.g., Linux) may use certain bits in the address to disambiguate between different types of addresses (e.g., kernel and user-mode addresses). Pointer masking *does not* take these semantics into account and is purely an arithmetic operation on the address it is given.



Linux places kernel addresses in the upper half of the address space and user addresses in the lower half of the address space. As such, the MSB is often used to identify the type of a particular address. With pointer masking enabled, this role is now played by bit XLEN-PMLEN-1 and code that checks whether a pointer is a kernel or a user address needs to inspect this bit instead. For backward compatibility, it may be desirable that the MSB still indicates whether an address is a user or a kernel address. An operating system's ABI may mandate this, but it does not affect the pointer masking mechanism itself. For example, the Linux ABI may choose to mandate that the MSB is not used for tagging and replicates bit XLEN-PMLEN-1 bit (note that for such a mechanism to be secure, the kernel needs to check the MSB of any user mode-supplied address and ensure that this invariant holds before using it; alternatively, it can apply the transformation from Listing 1 or 2 to ensure that the MSB is set to the correct value).

6.10.2.6. Memory Accesses Subject to Pointer Masking

Pointer masking applies to all explicit memory accesses. Currently, in the Base and Privileged ISAs, these are:

- **Base Instruction Set:** LB, LH, LW, LBU, LHU, LWU, LD, SB, SH, SW, SD.
- **Atomics:** All instructions in RV32A and RV64A.
- **Floating Point:** FLW, FLD, FLQ, FSW, FSD, FSQ.
- **Compressed:** All instructions mapping to any of the above, and C.LWSP, C.LDSP, C.FLWSP, C.FLDSP, C.SWSP, C.SDSP, C.FSWSP, C.FSDSP.
- **Hypervisor Extension:** HLV.*, HSV.* (in some cases; see [Section 5.2.1](#)).
- **Cache Management Operations:** All instructions in Zicbom, Zicbop and Zicboz.
- **Vector Extension:** All vector load and store instructions in the ratified RVV 1.0 spec.
- **Zicfiss Extension:** SSPUSH, C.SSPUSH, SSPOPCHK, C.SSPOPCHK, SSAMOSWAP.W/D.
- **Assorted:** FENCE, FENCE.I (if the currently unused address fields become enabled in the future).



This list will grow over time as new extensions introduce new instructions that perform explicit memory accesses.

For other extensions, pointer masking applies to all explicit memory accesses by default. Future extensions may add specific language to indicate whether particular accesses are or are not included in pointer masking.



*It is worth noting that pointer masking is not applied to **SFENCE.***, **HFENCE.***, **SINVAL.***, or **HINVAL.***. When such an operation is invoked, it is the responsibility of the software to provide*

the correct address.

MPRV and SPVP affect pointer masking as well, causing the pointer masking settings of the effective privilege mode to be applied. When MXR is in effect at the effective privilege mode where explicit memory access is performed, pointer masking does not apply.



Note that this includes cases where page-based virtual memory is not in effect; i.e., although MXR has no effect on permissions checks when page-based virtual memory is not in effect, it is still used in determining whether or not pointer masking should be applied.

Cache Management Operations (CMOs) must respect and take into account pointer masking. Otherwise, a few serious security problems can appear, including:



- *CBO.ZERO may work as a STORE operation. If pointer masking is not respected, it would be possible to write to memory bypassing the mask enforcement.*
- *If CMOs did not respect pointer masking, it would be possible to weaponize this in a side-channel attack. For example, U-mode would be able to flush a physical address (without masking) that it should not be permitted to.*

Pointer masking only applies to accesses generated by instructions on the CPU (including CPU extensions such as an FPU). E.g., it does not apply to accesses generated by page-table walks, the IOMMU, or devices.



Pointer Masking does not apply to DMA controllers and other devices. It is therefore the responsibility of the software to manually untag these addresses.

Misaligned accesses are supported, subject to the same limitations as in the absence of pointer masking. The behavior is identical to applying the pointer masking transformation to every constituent aligned memory access. In other words, the accessed bytes should be identical to the bytes that would be accessed if the pointer masking transformation was individually applied to every byte of the access without pointer masking. This ensures that both hardware implementations and emulation of misaligned accesses in M-mode behave the same way, and that the M-mode implementation is identical whether or not pointer masking is enabled (e.g., such an implementation may leverage MPRV to apply the correct privilege mode's pointer masking setting).

No pointer masking operations are applied when software reads/writes to CSRs, including those meant to hold addresses. If software stores tagged addresses into such CSRs, data load or data store operations based on those addresses are subject to pointer masking only if they are explicit (Section 6.10.2.6) and pointer masking is enabled for the privilege mode that performs the access. The implemented WARL width of CSRs is unaffected by pointer masking (e.g., if a CSR supports 52 bits of valid addresses and pointer masking is supported with PMLen=16, the necessary number of WARL bits remains 52 independently of whether pointer masking is enabled or disabled).

In contrast to software writes, pointer masking, when applicable, **is applied** for hardware writes to a CSR (e.g., when the hardware writes the transformed address to `stval` when taking an exception). Pointer masking is also applied, when applicable, to the memory access address when matching address triggers in debug.

For example, software is free to write a tagged or untagged address to `stvec`, but on trap delivery (e.g., due to an exception or interrupt), pointer masking **will not be applied** to the address of the trap handler. However, when delivering an exception, the hardware applies pointer masking to any address written into `stval` if pointer masking is applicable to that address.



The rationale for this choice is that delivering the additional bits may add overheads in some hardware implementations. Further, pointer masking is configured per privilege mode, so all trap handlers in supervisor mode would need to be careful to configure pointer masking the

same way as user mode or manually unmask (which is expensive).

6.10.2.7. Pointer Masking Extensions

Pointer masking refers to a number of separate extensions, all of which are privileged. This approach is used to capture optionality of pointer masking features. Profiles and implementations may choose to support an arbitrary subset of these extensions and must define valid ranges for their corresponding values of PMLen.

Extensions:

- **Ssnpm**: A supervisor-level extension that provides pointer masking for the next lower privilege mode (U-mode), and for VS- and VU-modes if the H extension is present. See [Section 4.1.10](#), [Section 5.2.5](#), [Section 5.2.1](#), and [Section 5.5.4](#).
- **Smpm**: A machine-level extension that provides pointer masking for the next lower privilege mode (S/HS if S-mode is implemented, or U-mode otherwise). See [Section 3.1.18](#).
- **Smmpm**: A machine-level extension that provides pointer masking for M-mode. See [Section 3.1.19](#).

In addition, the pointer masking standard defines two extensions that describe an execution environment but have no bearing on hardware implementations. These extensions are intended to be used in profile specifications where a User profile or a Supervisor profile can only reference User level or Supervisor level pointer masking functionality, and not the associated CSR controls that exist at a higher privilege level (i.e., in the execution environment).

- **Sspm**: An extension that indicates that there is pointer-masking support available in supervisor mode, with some facility provided in the supervisor execution environment to control pointer masking.
- **Supm**: An extension that indicates that there is pointer-masking support available in user mode, with some facility provided in the application execution environment to control pointer masking.

The precise nature of these facilities is left to the respective execution environment.

Pointer masking only applies to RV64. In RV32, trying to enable pointer masking will result in an illegal WARL write and not update the pointer masking configuration bits (see [Section 3.1.19](#), [Section 3.1.18](#), [Section 5.2.5](#), and [Section 4.1.10](#) for details). The same is the case on RV64 or larger systems when UXL/SXL/MXL is set to 1 for the corresponding privilege mode. Note that in RV32, the CSR bits introduced by pointer masking are still present, for compatibility between RV32 and larger systems with UXL/SXL/MXL set to 1. Setting UXL/SXL/MXL to 1 will clear the corresponding pointer masking configuration bits.



Note that setting UXL/SXL/MXL to 1 and back to 0 does not preserve the previous values of the PMM bits. This includes the case of entering an RV32 virtual machine from an RV64 hypervisor and returning.



Future extensions may introduce additional CSRs to allow different privilege modes to modify their own pointer masking settings. This may be required for future use cases in managed runtime systems that are not currently addressed as part of this extension.

6.10.2.8. Number of Masked Bits

As described in [Section 6.10.2.4](#), the supported values of PMLen may depend on the effective privilege mode. The current standard only defines PMLen=XLen-48 and PMLen=XLen-57, but this assumption may be relaxed in future extensions and profiles. Trying to enable pointer masking in an unsupported scenario represents an illegal write to the corresponding pointer masking enable bit and follows WARL

semantics. Future profiles may choose to define certain combinations of privilege modes and supported values of PMLEN as mandatory.



An option that was considered but discarded was to allow implementations to set PMLEN depending on the active addressing mode. For example, PMLEN could be set to 16 for Sv48 and to 25 for Sv39. However, having a single value of PMLEN (e.g., setting PMLEN to 16 for both Sv39 and Sv48 rather than 25) facilitates TLB implementations in designs that support Sv39 and Sv48 but not Sv57. 16 bits are sufficient for current pointer masking use cases but allow for a TLB implementation that matches against the same number of virtual tag bits independently of whether it is running with Sv39 or Sv48. However, if Sv57 is supported, tag matching may need to be conditional on the current address translation mode.

DRAFT

Chapter 7. "Sv" Supervisor Virtual-Memory Extensions



This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

7.1. "Svnapot" Extension for NAPOT Translation Contiguity, Version 1.0

In Sv39, Sv48, and Sv57, when a PTE has $N=1$, the PTE represents a translation that is part of a range of contiguous virtual-to-physical translations with the same values for PTE bits 5–0. Such ranges must be of a naturally aligned power-of-2 (NAPOT) granularity larger than the base page size.

The Svnapot extension depends on the Sv39 extension.

Table 156. Page table entry encodings when $pte.N=1$

i	$pte.ppn[i]$	Description	$pte.napot\_bits$
0	x xxxx xxx1	Reserved	-
0	x xxxx xx1x	Reserved	-
0	x xxxx x1xx	Reserved	-
0	x xxxx 1000	64 KiB contiguous region	4
0	x xxxx 0xxx	Reserved	-
≥ 1	x xxxx xxxx	Reserved	-

NAPOT PTEs behave identically to non-NAPOT PTEs within the address-translation algorithm in [Section 4.3.2](#), except that:

- If the encoding in pte is valid according to [Table 156](#), then instead of returning the original value of pte , implicit reads of a NAPOT PTE return a copy of pte in which $pte.ppn[i][pte.napot\_bits-1:0]$ is replaced by $vpn[i][pte.napot\_bits-1:0]$. If the encoding in pte is reserved according to [Table 156](#), then a page-fault exception must be raised.
- Implicit reads of NAPOT page table entries may create address-translation cache entries mapping $a + j \times PTESIZE$ to a copy of pte in which $pte.ppn[i][pte.napot\_bits-1:0]$ is replaced by $vpn[i][pte.napot\_bits-1:0]$, for any or all j such that $j \gg napot\_bits = vpn[i] \gg napot\_bits$, all for the address space identified in $satp$ as loaded by step 1.

The motivation for a NAPOT PTE is that it can be cached in a TLB as one or more entries representing the contiguous region as if it were a single (large) page covered by a single translation. This compaction can help relieve TLB pressure in some scenarios. The encoding is designed to fit within the pre-existing Sv39, Sv48, and Sv57 PTE formats so as not to disrupt existing implementations or designs that choose not to implement the scheme. It is also designed so as not to complicate the definition of the address-translation algorithm.



The address translation cache abstraction captures the behavior that would result from the creation of a single TLB entry covering the entire NAPOT region. It is also designed to be consistent with implementations that support NAPOT PTEs by splitting the NAPOT region into TLB entries covering any smaller power-of-two region sizes. For example, a 64 KiB NAPOT PTE might trigger the creation of 16 standard 4 KiB TLB entries, all with contents generated from the NAPOT PTE (even if the PTEs for the other 4 KiB regions have different contents).

In typical usage scenarios, NAPOT PTEs in the same region will have the same attributes, same PPNs, and same values for bits 5–0. RSW remains reserved for supervisor software control. It is the responsibility of the OS and/or hypervisor to configure the page tables in such a way that there are no inconsistencies between NAPOT PTEs and other NAPOT or non-

NAPOT PTEs that overlap the same address range. If an update needs to be made, the OS generally should first mark all of the PTEs invalid, then issue `SFENCE.VMA` instruction(s) covering all 4 KiB regions within the range (either via a single `SFENCE.VMA` with `rs1=x0`, or with multiple `SFENCE.VMA` instructions with `rs1≠x0`), then update the PTE(s), as described in [Section 4.2.1](#), unless any inconsistencies are known to be benign. If any inconsistencies do exist, then the effect is the same as when `SFENCE.VMA` is used incorrectly: one of the translations will be chosen, but the choice is unpredictable.

If an implementation chooses to use a NAPOT PTE (or cached version thereof), it might not consult the PTE directly specified by the algorithm in [Section 4.3.2](#) at all. Therefore, the `D` and `A` bits may not be identical across all mappings of the same address range even in typical use cases. The operating system must query all NAPOT aliases of a page to determine whether that page has been accessed and/or is dirty. If the OS manually sets the `A` and/or `D` bits for a page, it is recommended that the OS also set the `A` and/or `D` bits for other NAPOT aliases as appropriate in order to avoid unnecessary traps.

Just as with normal PTEs, TLBs are permitted to cache NAPOT PTEs whose `V` (Valid) bit is clear.

Depending on need, the NAPOT scheme may be extended to other intermediate page sizes and/or to other levels of the page table in the future. The encoding is designed to accommodate other NAPOT sizes should that need arise. For example:

i	pte.ppn[i]	Description	pte.napot_bits
0	x xxxx xxx1	8 KiB contiguous region	1
0	x xxxx xx10	16 KiB contiguous region	2
0	x xxxx x100	32 KiB contiguous region	3
0	x xxxx 1000	64 KiB contiguous region	4
0	x xxx1 0000	128 KiB contiguous region	5
...	...	...	...
1	x xxxx xxx1	4 MiB contiguous region	1
1	x xxxx xx10	8 MiB contiguous region	2
...	...	...	...

In such a case, an implementation may or may not support all options. The discoverability mechanism for this extension would be extended to allow system software to determine which sizes are supported.

Other sizes may remain deliberately excluded, so that PPN bits not being used to indicate a valid NAPOT region size (e.g., the least-significant bit of `pte.ppn[i]`) may be repurposed for other uses in the future.

However, in case finer-grained intermediate page size support proves not to be useful, we have chosen to standardize only 64 KiB support as a first step.

If the hypervisor extension is also implemented, Svnapot is also supported in G-stage translation.

7.2. "Svpbmt" Extension for Page-Based Memory Types, Version 1.0

In Sv39, Sv48, and Sv57, bits 62-61 of a leaf page table entry indicate the use of page-based memory types that override the PMA(s) for the associated memory pages. The encoding for the PBMT bits is captured in

Table 157.

The Svpbmt extension depends on the Sv39 extension.

Table 157. Encodings for PBMT field in Sv39, Sv48, and Sv57 PTEs.

Mode	Value	Requested Memory Attributes
PMA	0	None
NC	1	Non-cacheable, idempotent, weakly-ordered (RVWMO), main memory
IO	2	Non-cacheable, non-idempotent, strongly-ordered (I/O ordering), I/O
-	3	<i>Reserved for future standard use</i>

Implementations may override additional PMAs not explicitly listed in Table 157. For example, to be consistent with the characteristics of a typical I/O region, a misaligned memory access to a page with PBMT=IO might raise an exception, even if the underlying region were main memory and the same access would have succeeded for PBMT=PMA.



Future extensions may provide more and/or finer-grained control over which PMAs can be overridden.

For non-leaf PTEs, bits 62-61 are reserved for future standard use. Until their use is defined by a standard extension, they must be cleared by software for forward compatibility, or else a page-fault exception is raised.

For leaf PTEs, setting bits 62-61 to the value 3 is reserved for future standard use. Until this value is defined by a standard extension, using this reserved value in a leaf PTE raises a page-fault exception.

When PBMT settings override a main memory page into I/O or vice versa, memory accesses to such pages obey the memory ordering rules of the final effective attribute, as follows.

If the underlying physical memory attribute for a page is I/O, and the page has PBMT=NC, then accesses to that page obey RVWMO. However, accesses to such pages are considered to be *both* I/O and main memory accesses for the purposes of FENCE, .aq, and .rl.

If the underlying physical memory attribute for a page is main memory, and the page has PBMT=IO, then accesses to that page obey strong channel O I/O ordering rules. However, accesses to such pages are considered to be *both* I/O and main memory accesses for the purposes of FENCE, .aq, and .rl.



A device driver written to rely on I/O strong ordering rules will not operate correctly if the address range is mapped with PBMT=NC. As such, this configuration is discouraged.

It will often still be useful to map physical I/O regions using PBMT=NC so that write combining and speculative accesses can be performed. Such optimizations will likely improve performance when applied with adequate care.

When Svpbmt is used with non-zero PBMT encodings, it is possible for multiple virtual aliases of the same physical page to exist simultaneously with different memory attributes. It is also possible for a U-mode or S-mode mapping through a PTE with Svpbmt enabled to observe different memory attributes for a given region of physical memory than a concurrent access to the same page performed by M-mode or when MODE=Bare. In such cases, the behaviors dictated by the attributes (including coherence, which is otherwise unaffected) may be violated.

Accessing the same location using different attributes that are both non-cacheable (e.g., NC and IO) does not cause loss of coherence, but might result in weaker memory ordering than the stricter attribute ordinarily guarantees. Executing a **fence iorw**, **iorw** instruction between such accesses suffices to prevent loss of memory ordering.

Accessing the same location using different cacheability attributes may cause loss of coherence. Executing the following sequence between such accesses prevents both loss of coherence and loss of memory ordering: **fence iorw**, **iorw**, followed by **cbo.flush** to an address of that location, followed by a **fence iorw**, **iorw**.

It follows that, if the same location might later be referenced using the original attributes, then this sequence must be repeated beforehand.



In certain cases, a weaker sequence might suffice to prevent loss of coherence. These situations will be detailed following the forthcoming formalization of the interaction of the RVWMO memory model with the instructions in the Zicbom extension.

When two-stage address translation is enabled within the H extension, the page-based memory types are also applied in two stages. First, if **hgatp.MODE** is not equal to zero, non-zero G-stage PTE PBMT bits override the attributes in the PMA to produce an intermediate set of attributes. Otherwise, the PMAs serve as the intermediate attributes. Second, if **vsatp.MODE** is not equal to zero, non-zero VS-stage PTE PBMT bits override the intermediate attributes to produce the final set of attributes used by accesses to the page in question. Otherwise, the intermediate attributes are used as the final set of attributes.



These final attributes apply to implicit and explicit accesses that are subject to both stages of address translation. For accesses that are not subject to the first stage of address translation, e.g. VS-stage page-table accesses, the intermediate attributes apply instead.

7.3. "Svadu" Extension for Hardware Updating of A/D Bits, Version 1.0

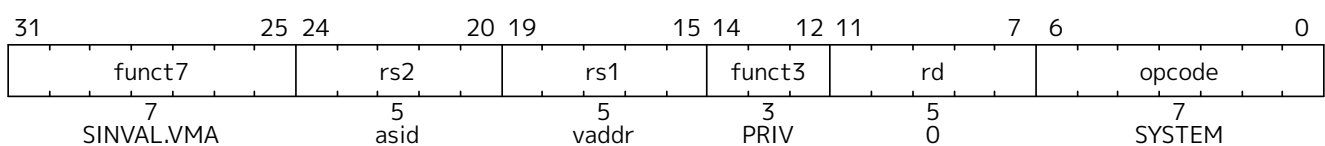
The Svadu extension adds support and CSR controls for hardware updating of PTE A/D bits.

If the Svadu extension is implemented, the **menvcfg.ADUE** field is writable. If the hypervisor extension is additionally implemented, the **henvcfg.ADUE** field is also writable. See [Section 3.1.18](#) and [Section 5.2.5](#) for the definitions of those fields.

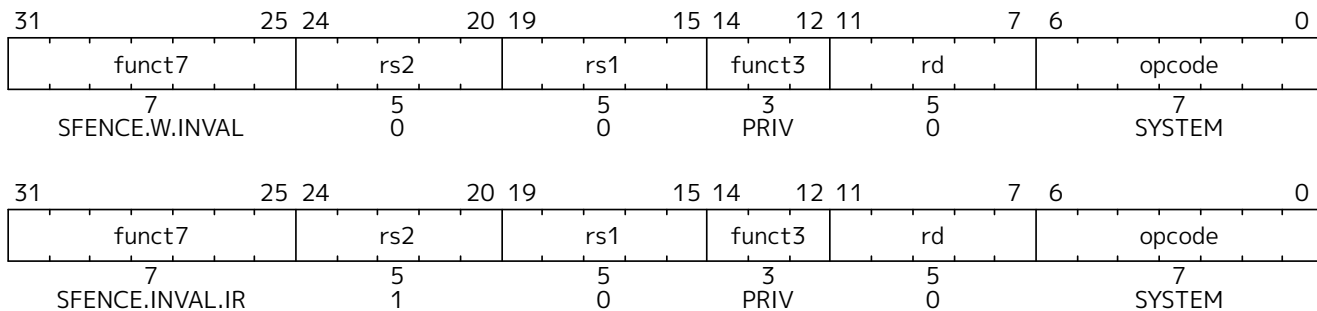
[Section 4.3.1](#) defines the semantics of hardware updating of A/D bits. When hardware updating of A/D bits is disabled, the Svade extension, which mandates exceptions when A/D bits need be set, instead takes effect. The Svade extension is also defined in [Section 4.3.1](#).

7.4. "Svinval" Extension for Fine-Grained Address-Translation Cache Invalidation, Version 1.0

The Svinval extension splits **SFENCE.VMA**, **HFENCE.VVMA**, and **HFENCE.GVMA** instructions into finer-grained invalidation and ordering operations that can be more efficiently batched or pipelined on certain classes of high-performance implementation.



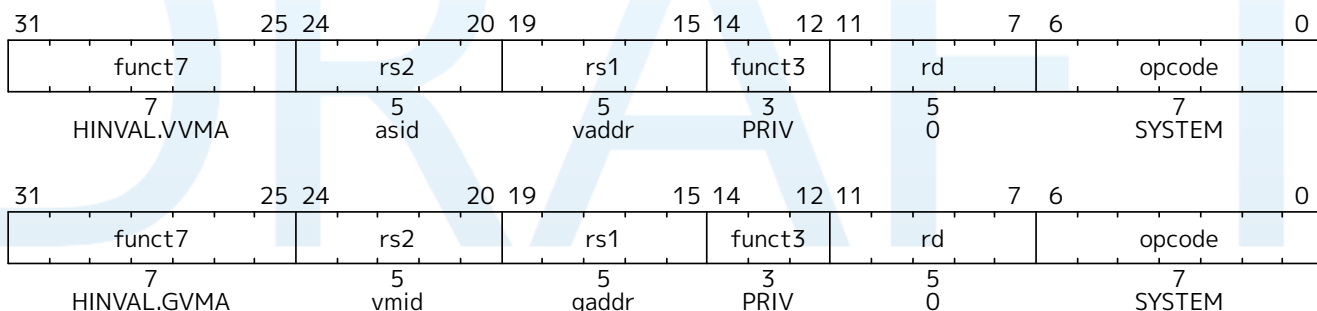
The **SINVAL.VMA** instruction invalidates any address-translation cache entries that an **SFENCE.VMA** instruction with the same values of **rs1** and **rs2** would invalidate. However, unlike **SFENCE.VMA**, **SINVAL.VMA** instructions are only ordered with respect to **SFENCE.VMA**, **SFENCE.W.INVALID**, and **SFENCE.INVALID.IR** instructions as defined below.



The SFENCE.W.INVALID instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent SINVAL.VMA instructions executed by the same hart. The SFENCE.INVALID.IR instruction guarantees that any previous SINVAL.VMA instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

When executed in order (but not necessarily consecutively) by a single hart, the sequence SFENCE.W.INVALID, SINVAL.VMA, and SFENCE.INVALID.IR has the same effect as a hypothetical SFENCE.VMA instruction in which:

- the values of *rs1* and *rs2* for the SFENCE.VMA are the same as those used in the SINVAL.VMA,
- reads and writes prior to the SFENCE.W.INVALID are considered to be those prior to the SFENCE.VMA, and
- reads and writes following the SFENCE.INVALID.IR are considered to be those subsequent to the SFENCE.VMA.



If the hypervisor extension is implemented, the Svinval extension also provides two additional instructions: HINVAL.VVMA and HINVAL.GVMA. These have the same semantics as SINVAL.VMA, except that they combine with SFENCE.W.INVALID and SFENCE.INVALID.IR to replace HFENCE.VVMA and HFENCE.GVMA, respectively, instead of SFENCE.VMA. In addition, HINVAL.GVMA uses VMIDs instead of ASIDs.

SINVAL.VMA, HINVAL.VVMA, and HINVAL.GVMA require the same permissions and raise the same exceptions as SFENCE.VMA, HFENCE.VVMA, and HFENCE.GVMA, respectively. In particular, an attempt to execute any of these instructions in U-mode always raises an illegal-instruction exception. An attempt to execute SINVAL.VMA or HINVAL.GVMA in S-mode or HS-mode when **mstatus.TVM**=1 also raises an illegal-instruction exception. An attempt to execute HINVAL.VVMA or HINVAL.GVMA in VS-mode or VU-mode, or to execute SINVAL.VMA in VU-mode, raises a virtual-instruction exception. When **hstatus.VTVM**=1, an attempt to execute SINVAL.VMA in VS-mode also raises a virtual-instruction exception.

Attempting to execute SFENCE.W.INVALID or SFENCE.INVALID.IR in U-mode raises an illegal-instruction exception. Doing so in VU-mode raises a virtual-instruction exception. SFENCE.W.INVALID and SFENCE.INVALID.IR are unaffected by the **mstatus.TVM** and **hstatus.VTVM** fields and hence are always permitted in S-mode and VS-mode.



SFENCE.W.INVALID and SFENCE.INVALID.IR instructions do not need to be trapped when `mstatus.TVM=1` or when `hstatus.VTVM=1`, as they only have ordering effects but no visible side effects. Trapping of the `SINVAL.VMA` instruction is sufficient to enable emulation of the intended overall TLB maintenance functionality.

In typical usage, software will invalidate a range of virtual addresses in the address-translation caches by executing an `SFENCE.W.INVALID` instruction, executing a series of `SINVAL.VMA`, `HINVAL.VVMA`, or `HINVAL.GVMA` instructions to the addresses (and optionally ASIDs or VMIDs) in question, and then executing an `SFENCE.INVALID.IR` instruction.

High-performance implementations will be able to pipeline the address-translation cache invalidation operations, and will defer any pipeline stalls or other memory ordering enforcement until an `SFENCE.W.INVALID`, `SFENCE.INVALID.IR`, `SFENCE.VMA`, `HFENCE.GVMA`, or `HFENCE.VVMA` instruction is executed.

Simpler implementations may implement `SINVAL.VMA`, `HINVAL.VVMA`, and `HINVAL.GVMA` identically to `SFENCE.VMA`, `HFENCE.VVMA`, and `HFENCE.GVMA`, respectively, while implementing `SFENCE.W.INVALID` and `SFENCE.INVALID.IR` instructions as no-ops.

7.5. "Svtpc" Extension for Obviating Memory-Management Instructions after Marking PTEs Valid, Version 1.0

When the Svtpc extension is implemented, explicit stores by a hart that update the Valid bit of leaf and/or non-leaf PTEs from 0 to 1 and are visible to a hart will eventually become visible within a bounded timeframe to subsequent implicit accesses by that hart to such PTEs.



Svtpc relieves an operating system from executing certain memory-management instructions, such as `SFENCE.VMA` or `SINVAL.VMA`, which would normally be used to synchronize the hart's address-translation caches when a memory-resident PTE is changed from Invalid to Valid. Synchronizing the hart's address-translation caches with other forms of updates to a memory-resident PTE, including when a PTE is changed from Valid to Invalid, requires the use of suitable memory-management instructions. Svtpc guarantees that a change to a PTE from Invalid to Valid is made visible within a bounded time, thereby making the execution of these memory-management instructions redundant. The performance benefit of eliding these instructions outweighs the cost of an occasional gratuitous additional page fault that may occur.

Depending on the microarchitecture, some possible ways to facilitate implementation of Svtpc include: not having any address-translation caches, not storing Invalid PTEs in the address-translation caches, automatically evicting Invalid PTEs using a bounded timer, or making address-translation caches coherent with store instructions that modify PTEs.

7.6. "Svrsw60t59b" Extension for PTE Reserved-for-Software Bits 60-59, Version 1.0

If the Svrsw60t59b extension is implemented, then bits 60-59 of the page table entries (PTEs) are reserved for use by supervisor software and are ignored by the implementation.

If the Hypervisor (H) extension is also implemented, then bits 60-59 of the G-stage PTEs are reserved for use by supervisor software and are ignored by the implementation.

The Svrsw60t59b extension depends on Sv39.



Operating systems frequently use reserved bits within PTEs to store metadata for advanced memory management features. Embedding these metadata bits directly within the PTEs allows for fast access with minimal overhead, avoiding costly lookups in auxiliary data structures. By default, Sv39 and Sv39x4 require a page fault and a guest-page fault exception, respectively, to be raised if bits 60–59 are not zero.

DRAFT

Chapter 8. "Ss" Supervisor Extensions



This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

8.1. "Ssqosid" Extension for Quality-of-Service (QoS) Identifiers, Version 1.0

Quality of Service (QoS) is defined as the minimal end-to-end performance guaranteed in advance by a service level agreement (SLA) to a workload. Performance metrics might include measures such as instructions per cycle (IPC), latency of service, etc.

When multiple workloads execute concurrently on modern processors—equipped with large core counts, multiple cache hierarchies, and multiple memory controllers— the performance of any given workload becomes less deterministic, or even non-deterministic, due to shared resource contention.

To manage performance variability, system software needs resource allocation and monitoring capabilities. These capabilities allow for the reservation of resources like cache and bandwidth, thus meeting individual performance targets while minimizing interference. For resource management, hardware should provide monitoring features that allow system software to profile workload resource consumption and allocate resources accordingly.

To facilitate this, the QoS Identifiers extension (Ssqosid) introduces the **srmcfg** register, which configures a hart with two identifiers: a Resource Control ID (**RCID**) and a Monitoring Counter ID (**MCID**). These identifiers accompany each request issued by the hart to shared resource controllers.

Additional metadata, like the nature of the memory access and the ID of the originating supervisor domain, can accompany **RCID** and **MCID**. Resource controllers may use this metadata for differentiated service such as a different capacity allocation for code storage vs. data storage. Resource controllers can use this data for security policies such as not exposing statistics of one security domain to another.

These identifiers are crucial for the RISC-V Capacity and Bandwidth Controller QoS Register Interface (CBQRI) specification, which provides methods for setting resource usage limits and monitoring resource consumption. The **RCID** controls resource allocations, while the **MCID** is used for tracking resource usage.



The Ssqosid extension does not require that S-mode mode be implemented.

8.1.1. Supervisor Resource Management Configuration (**srmcfg**) register

The **srmcfg** register is an SXLEN-bit read/write register used to configure a Resource Control ID (**RCID**) and a Monitoring Counter ID (**MCID**). Both **RCID** and **MCID** are WARL fields. The register is formatted as shown in [Figure 130](#) when SXLEN=64 and [Figure 131](#) when SXLEN=32.

The **RCID** and **MCID** accompany each request made by the hart to shared resource controllers. The **RCID** is used to determine the resource allocations (e.g., cache occupancy limits, memory bandwidth limits, etc.) to enforce. The **MCID** is used to identify a counter to monitor resource usage.

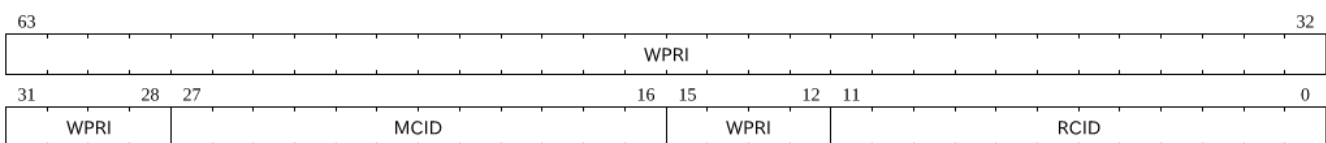


Figure 130. Supervisor Resource Management Configuration (**srmcfg**) register for SXLEN=64

8.4. Sstvecd Extension for Direct Trap Vectoring, Version 1.0

If the Sstvecd extension is implemented, then `stvec.MODE` must be capable of holding the value 0 (Direct). Furthermore, when `stvec.MODE=Direct`, `stvec.BASE` must be capable of holding any valid four-byte-aligned address.

8.5. Sstvala Extension for Trap Value Reporting, Version 1.0

If the Sstvala extension is implemented, then `stval` must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions that are defined to write an address to `stval`, other than those caused by execution of the `EBREAK` or `C.EBREAK` instructions. For virtual-instruction and illegal-instruction exceptions, `stval` must be written with the faulting instruction.

8.6. Sscounterenw Extension for Counter-Enable Writability, Version 1.0

If the Sscounterenw extension is implemented, then for any `hpmcounter` that is not read-only zero, the corresponding bit in `scounteren` must be writable.

8.7. Ssstrict Extension for Extension Conformance, Version 1.0

If the Ssstrict extension is implemented, then no non-conforming extensions are present. Furthermore, attempts to execute unimplemented opcodes or access unimplemented CSRs in the standard or reserved encoding spaces raises an illegal instruction exception that results in a contained trap to the supervisor-mode trap handler.

8.8. "Sstc" Extension for Supervisor-mode Timer Interrupts, Version 1.0

The current Privileged arch specification only defines a hardware mechanism for generating machine-mode timer interrupts (based on the `mtime` and `mtimecmp` registers). With the resultant requirement that timer services for S-mode/HS-mode (and for VS-mode) have to all be provided by M-mode - via SBI calls from S/HS-mode up to M-mode (or VS-mode calls to HS-mode and then to M-mode). M-mode software then multiplexes these multiple logical timers onto its one physical M-mode timer facility, and the M-mode timer interrupt handler passes timer interrupts back down to the appropriate lower privilege mode.

This extension serves to provide supervisor mode with its own CSR-based timer interrupt facility that it can directly manage to provide its own timer service (in the form of having its own `stimecmp` register) - thus eliminating the large overheads for emulating S/HS-mode timers and timer interrupt generation up in M-mode. Further, this extension adds a similar facility to the Hypervisor extension for VS-mode.

The extension name is "Sstc" ('Ss' for Privileged arch and Supervisor-level extensions, and 'tc' for `timecmp`). This extension adds the S-level `stimecmp` CSR ([Section 4.1.12](#)) and the VS-level `vstimecmp` CSR ([Section 5.2.19](#)). This extension adds the STCE bit to the `menvcfg` ([Section 3.1.18](#)) and `henvcfg` ([Section 5.2.5](#)) CSRs.

8.9. "Sscofpmf" Extension for Count Overflow and Mode-Based Filtering, Version 1.0

The current Privileged specification defines `mhpmevent` CSRs to select and control event counting by the associated `hpmcounter` CSRs, but provides no standardization of any fields within these CSRs. For at least Linux-class rich-OS systems it is desirable to standardize certain basic features that are broadly desired

(and have come up over the past year plus on RISC-V lists, as well as have been the subject of past proposals). This enables there to be standard upstream software support that eliminates the need for implementations to provide their own custom software support.

This extension serves to accomplish exactly this within the existing mhpmevent CSRs (and correspondingly avoids the unnecessary creation of whole new sets of CSRs - past just one new CSR).

This extension sticks to addressing two basic well-understood needs that have been requested by various people. To make it easy to understand the deltas from the current Priv 1.11/1.12 specs, this is written as the actual exact changes to be made to existing paragraphs of Priv spec text (or additional paragraphs within the existing text).

The extension name is "Sscofpmf" ('Ss' for Privileged arch and Supervisor-level extensions, and 'cofpmf' for Count OverFlow and Privilege Mode Filtering).

Note that the new count overflow interrupt will be treated as a standard local interrupt that is assigned to bit 13 in the mip/mie/sip/sie registers.

8.9.1. Count Overflow Control

The following bits are added to mhpmevent:

63	62	61	60	59	58	57	56
OF	MINH	SINH	UINH	VSINH	VUINH	WPRI	WPRI

Field	Description
OF	Overflow status and interrupt disable bit that is set when counter overflows
MINH	If set, then counting of events in M-mode is inhibited
SINH	If set, then counting of events in S/HS-mode is inhibited
UINH	If set, then counting of events in U-mode is inhibited
VSINH	If set, then counting of events in VS-mode is inhibited
VUINH	If set, then counting of events in VU-mode is inhibited
WPRI	Reserved
WPRI	Reserved

For each xINH bit, if the associated privilege mode is not implemented, the bit is read-only zero.

Each of the five xINH bits, when set, inhibit counting of events while in privilege mode x. All-zeroes for these bits results in counting of events in all modes.

The OF bit is set when the corresponding hpmcounter overflows, and remains set until written by software. Since hpmcounter values are unsigned values, overflow is defined as unsigned overflow of the implemented counter bits. Note that there is no loss of information after an overflow since the counter wraps around and keeps counting while the sticky OF bit remains set.

If supervisor mode is implemented, the 32-bit scountovf register contains read-only shadow copies of the OF bits in all 29 mhpmevent registers.

If an hpmcounter overflows while the associated OF bit is zero, then a "count overflow interrupt request" is generated. If the OF bit is one, then no interrupt request is generated. Consequently the OF bit also functions as a count overflow interrupt disable for the associated hpmcounter.

Count overflow never results from writes to the `mhpmcountern` or `mhpmeventn` registers, only from hardware increments of counter registers.

This count-overflow-interrupt-request signal is treated as a standard local interrupt that corresponds to bit 13 in the `mip/mie/sip/sie` registers. The `mip/sip` LCOFIP and `mie/sie` LCOFIE bits are, respectively, the interrupt-pending and interrupt-enable bits for this interrupt. ('LCOFI' represents 'Local Count Overflow Interrupt'.)

Generation of a count-overflow-interrupt request by an `hpmcounter` sets the associated OF bit. When an OF bit is set, it eventually, but not necessarily immediately, sets the LCOFIP bit in the `mip/sip` registers. The LCOFIP bit is cleared by software before servicing the count overflow interrupt resulting from one or more count overflows. The `midelleg` register controls the delegation of this interrupt to S-mode versus M-mode.#



There are not separate overflow status and overflow interrupt enable bits. In practice, enabling overflow interrupt generation (by clearing the OF bit) is done in conjunction with initializing the counter to a starting value. Once a counter has overflowed, it and the OF bit must be reinitialized before another overflow interrupt can be generated.



Software can distinguish newly overflowed counters (yet to be serviced by an overflow interrupt handler) from overflowed counters that have already been serviced or that are configured to not generate an interrupt on overflow, by maintaining a bit mask reflecting which counters are active and due to eventually overflow.

8.9.2. Supervisor Count Overflow (`scountovf`) Register

This extension adds the `scountovf` CSR, a 32-bit read-only register that contains shadow copies of the OF bits in the 29 `mhpmevent` CSRs (`mhpmevent3` - `mhpmevent31`) - where `scountovf` bit *X* corresponds to `mhpmeventX`.

This register enables supervisor-level overflow interrupt handler software to quickly and easily determine which counter(s) have overflowed (without needing to make an execution environment call or series of calls ultimately up to M-mode).

Read access to bit *X* is subject to the same `mcounteren` (or `mcounteren` and `hcounteren`) CSRs that mediate access to the `hpmcounter` CSRs by S-mode (or VS-mode). In M-mode, `scountovf` bit *X* is always readable. In S/HS-mode, `scountovf` bit *X* is readable when `mcounteren` bit *X* is set, and otherwise reads as zero. Similarly, in VS mode, `scountovf` bit *X* is readable when `mcounteren` bit *X* and `hcounteren` bit *X* are both set, and otherwise reads as zero.

8.10. "Ssdbltrap" Double Trap Extension, Version 1.0

The `Ssdbltrap` extension addresses a double trap (See [Section 3.1.6.2](#)) privilege modes lower than M. It enables HS-mode to invoke a critical error handler in a virtual machine on a double trap in VS-mode. It also allows M-mode to invoke a critical error handler in the OS/Hypervisor on a double trap in S/HS-mode.

The `Ssdbltrap` extension adds the `menvcfg.DTE` (See [Section 3.1.18](#)) and the `sstatus.SDT` fields (See [Section 4.1.1](#)). If the hypervisor extension is additionally implemented, then the extension adds the `henvcfg.DTE` (See [Section 5.2.5](#)) and the `vsstatus.SDT` fields (See [Section 5.2.11](#)).

See [Section 4.1.1.5](#) for the operational details.

Chapter 9. "Sh" Hypervisor Extensions



This chapter is currently being restructured. Its contents are normative, but the presentation might appear disjoint.

9.1. Shvstvecd Extension for Direct Trap Vectoring, Version 1.0

If the Shvstvecd extension is implemented, then `vstvec.MODE` must be capable of holding the value 0 (Direct). Furthermore, when `vstvec.MODE`=Direct, `vstvec.BASE` must be capable of holding any valid four-byte-aligned address.

9.2. Shcounterenw Extension for Counter-Enable Writability, Version 1.0

If the Shcounterenw extension is implemented, then for any `hpmcounter` that is not read-only zero, the corresponding bit in `hcounteren` must be writable.

9.3. Shvstvala Extension for Trap Value Reporting, Version 1.0

If the Shvstvala extension is implemented, `vstval` must be written in all cases described in [Section 8.5](#) for `stval`.

9.4. Shtvala Extension for Trap Value Reporting, Version 1.0

If the Shtvala extension is implemented, `htval` must be written with the faulting guest physical address in all circumstances permitted by the ISA.

9.5. Shvsatpa Extension for Translation Mode Support, Version 1.0

If the Shvsatpa extension is implemented, all translation modes supported in `satp` must be supported in `vsatp`.

9.6. Shgatpa Extension for Translation Mode Support, Version 1.0

If the Shgatpa extension is implemented, then for each supported virtual memory scheme SvNN supported in `satp`, the corresponding `hgatp` SvNNx4 mode must be supported. Furthermore, the `hgatp` mode Bare must also be supported.

9.7. Sha Augmented Hypervisor Extension

The Augmented Hypervisor Extension, Sha, adds several minor features to the hypervisor extension. It depends on the following extensions:

- [H](#)
- [Ssstateen](#)
- [Shcounterenw](#)
- [Shvstvala](#)
- [Shtvala](#)

- [Shvstvecd](#)
- [Shvsatpa](#)
- [Shgatpa](#)

DRAFT

Chapter 10. RISC-V Privileged Instruction Set Listings

This chapter presents instruction-set listings for all instructions defined in the RISC-V Privileged Architecture.

The instruction-set listings for unprivileged instructions, including the ECALL and EBREAK instructions, are provided in [Volume I, Chapter 34](#).

DRAFT

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

Trap-Return Instructions

0001000	00010	00000	000	00000	1110011	SRET
0011000	00010	00000	000	00000	1110011	MRET
0111000	00010	00000	000	00000	1110011	MNRET

Interrupt-Management Instructions

0001000	00101	00000	000	00000	1110011	WFI

Control Transfer Records Management Instructions

0001000	00100	00000	000	00000	1110011	SCTRCLR

Supervisor Memory-Management Instructions

0001001	rs2	rs1	000	00000	1110011	SFENCE.VMA

Hypervisor Memory-Management Instructions

0010001	rs2	rs1	000	00000	1110011	HFENCE.VVMA
0110001	rs2	rs1	000	00000	1110011	HFENCE.GVMA

Hypervisor Virtual-Machine Load and Store Instructions

0110000	00000	rs1	100	rd	1110011	HLV.B
0110000	00001	rs1	100	rd	1110011	HLV.BU
0110010	00000	rs1	100	rd	1110011	HLV.H
0110010	00001	rs1	100	rd	1110011	HLV.HU
0110100	00000	rs1	100	rd	1110011	HLV.W
0110010	00011	rs1	100	rd	1110011	HLVX.HU
0110100	00011	rs1	100	rd	1110011	HLVX.WU
0110001	rs2	rs1	100	00000	1110011	HSV.B
0110011	rs2	rs1	100	00000	1110011	HSV.H
0110101	rs2	rs1	100	00000	1110011	HSV.W

Hypervisor Virtual-Machine Load and Store Instructions, RV64 only

0110100	00001	rs1	100	rd	1110011	HLV.WU
0110110	00000	rs1	100	rd	1110011	HLV.D
0110111	rs2	rs1	100	00000	1110011	HSV.D

Svinval Memory-Management Extension

0001011	rs2	rs1	000	00000	1110011	SINVAL.VMA
0001100	00000	00000	000	00000	1110011	SFENCE.W.INVALID
0001100	00001	00000	000	00000	1110011	SFENCE.INVALID.IR
0010011	rs2	rs1	000	00000	1110011	HINVAL.VVMA
0110011	rs2	rs1	000	00000	1110011	HINVAL.GVMA

Figure 132. RISC-V Privileged Instructions

Appendix A: 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.

A.1. "Smepmp" Extension for PMP Enhancements for memory access and execution prevention in Machine mode

1. Since a CSR for security and / or global PMP behavior settings is not available with the current spec, we needed to define a new `mseccfg` CSR. This new CSR will allow us to add further security configuration options in the future and also allow developers to verify the existence of the new mechanisms defined on this extension.
2. There are use cases where developers want to enforce PMP rules in M-mode during the boot process, that are also able to modify, merge, and / or remove later on. Since a rule that is enforced in M-mode also needs to be locked (or else badly written or malicious M-mode software can remove it at any time), the only way for developers to approach this is to keep adding PMP rules to the chain and rely on rule priority. This is a waste of PMP rules and since it's only needed during boot, `mseccfg.RLB` is a simple workaround that can be used temporarily and then disabled and locked down.

Also when `mseccfg.MML` is set, according to 4b it's not possible to add a *Shared-Region* rule with executable privileges. So RLB can be set temporarily during the boot process to register such regions. Note that it's still possible to register executable *Shared-Region* rules using initial register settings (that may include `mseccfg.MML` being set and the rule being set on PMP registers) on **PMP reset**, without using RLB.



Be aware that RLB introduces a security vulnerability if left set after the boot process is over and in general it should be used with caution, even when used temporarily. Having editable PMP rules in M-mode gives a false sense of security since it only takes a few malicious instructions to lift any PMP restrictions this way. It doesn't make sense to have a security control in place and leave it unprotected. Rule Locking Bypass is only meant as a way to optimize the allocation of PMP rules, catch errors during debugging, and allow the bootrom/firmware to register executable *Shared-Region* rules. If developers / vendors have no use for such functionality, they should never set `mseccfg.RLB` and if possible hard-wire it to 0. In any case RLB should be disabled and locked as soon as possible.



If `mseccfg.RLB` is not used and left unset, it will be locked as soon as a PMP rule/entry with the `pmppcfg.L` bit set is configured.



Since PMP rules with a higher priority override rules with a lower priority, locked rules must precede non-locked rules.

3. With the current spec M-mode can access any memory region unless restricted by a PMP rule with the `pmppcfg.L` bit set. There are cases where this approach is overly permissive, and although it's possible to restrict M-mode by adding PMP rules during the boot process, this can also be seen as a waste of PMP rules. Having the option to block anything by default, and use PMP as an allowlist for M-mode is considered a safer approach. This functionality may be used during the boot process or upon **PMP reset**, using initial register settings.
4. The current dual meaning of the `pmppcfg.L` bit that marks a rule as Locked and **enforced** on all modes is

neither flexible nor clean. With the introduction of *Machine Mode Lock-down* the `pmpcfg.L` bit distinguishes between rules that are **enforced only** in M-mode (*M-mode-only*) or **only** in S/U-modes (*S/U-mode-only*). The rule locking becomes part of the definition of an *M-mode-only* rule, since when a rule is added in M mode, if not locked, can be modified or removed in a few instructions. On the other hand, S/U modes can't modify PMP rules anyway so locking them doesn't make sense.

- a. This separation between *M-mode-only* and *S/U-mode-only* rules also allows us to distinguish which regions are to be used by processes in Machine mode (`pmpcfg.L == 1`) and which by Supervisor or User mode processes (`pmpcfg.L == 0`), in the same way the U bit on the Virtual Memory's PTEs marks which Virtual Memory pages are to be used by User mode applications (U=1) and which by the Supervisor / OS (U=0). With this distinction in place we are able to implement memory access and execution prevention in M-mode for any physical memory region that is not *M-mode-only*.

An attacker that manages to tamper with a memory region used by S/U mode, even after successfully tricking a process running in M-mode to use or execute that region, will fail to perform a successful attack since that region will be *S/U-mode-only* hence any access when in M-mode will trigger an access exception.



In order to support zero-copy transfers between M-mode and S/U-mode we need to either allow shared memory regions, or introduce a mechanism similar to the `sstatus.SUM` bit to temporarily allow the high-privileged mode (in this case M-mode) to be able to perform loads and stores on the region of a less-privileged process (in this case S/U-mode). In our case after discussion within the group it seemed a better idea to follow the first approach and have this functionality encoded on a per-rule basis to avoid the risk of leaving a temporary, global bypass active when exiting M-mode, hence rendering memory access prevention useless.



*Although it's possible to use `mstatus.MPRV` in M-mode to read/write data on an S/U-mode-only region using general purpose registers for copying, this will happen with S/U-mode permissions, honoring any MMU restrictions put in place by S-mode. Of course it's still possible for M-mode to tamper with the page tables and / or add S/U-mode-only rules and bypass the protections put in place by S-mode but if an attacker has managed to compromise M-mode to such extent, no security guarantees are possible in any way. Also note that the threat model we present here assumes **buggy software in M-mode, not compromised software**. We considered disabling `mstatus.MPRV` but it seemed too much and out of scope.*

Shared-region rules can be used both for zero-copy data transfers and for sharing code segments. The latter may be used for example to allow S/U-mode to execute code by the vendor, that makes use of some vendor-specific ISA extension, without having to go through the firmware with an ecall. This is similar to the vDSO approach followed on Linux, that allows user space code to execute kernel code without having to perform a system call.

To make sure that shared data regions can't be executed and shared code regions can't be modified, the encoding changes the meaning of the `pmpcfg.X` bit. In case of shared data regions, with the exception of the `pmpcfg.LRWX=1111` encoding, the `pmpcfg.X` bit marks the capability of S/U-mode to write to that region, so it's not possible to encode an executable shared data region. In case of shared code regions, the `pmpcfg.X` bit marks the capability of M-mode to read from that region, and since `pmpcfg.RW=01` is used for encoding the shared region, it's not possible to encode a shared writable code region.



For adding Shared-region rules with executable privileges to share code segments between M-mode and S/U-mode, `mseccfg.RLB` needs to be implemented, or else such rules can only be added together with `mseccfg.MML` being set on PMP Reset. That's because the reserved encoding `pmpcfg.RW=01` being used for Shared-region rules is

only defined when `mseccfg.MML` is set, and 4b prevents the addition of rules with executable privileges on M-mode after `mseccfg.MML` is set unless `mseccfg.RLB` is also set.



Using the `pmppcfg.LRWX=1111` encoding for a locked shared read-only data region was decided later on, its initial meaning was an M-mode-only read/write/execute region. The reason for that change was that the already defined shared data regions were not locked, so r/w access to M-mode couldn't be restricted. In the same way we have execute-only shared code regions for both modes, it was decided to also be able to allow a least-privileged shared data region for both modes. This approach allows for example to share the `.text` section of an ELF with a shared code region and the `.rodata` section with a locked shared data region, without allowing M-mode to modify `.rodata`. We also decided that having a locked read/write/execute region in M-mode doesn't make much sense and could be dangerous, since M-mode won't be able to add further restrictions there (as in the case of S/U-mode where S-mode can further limit access to an `pmppcfg.LWRX=0111` region through the MMU), leaving the possibility of modifying an executable region in M-mode open.



For encoding Shared-region rules initially we used one of the two reserved bits on `pmppcfg` (bit 5) but in order to avoid allocating an extra bit, since those bits are a very limited resource, it was decided to use the reserved `R=0,W=1` combination.

- b. The idea with this restriction is that after the Firmware or the OS running in M-mode is initialized and `mseccfg.MML` is set, no new code regions are expected to be added since nothing else is expected to run in M-mode (everything else will run in S/U mode). Since we want to limit the attack surface of the system as much as possible, it makes sense to disallow any new code regions which may include malicious code, to be added/executed in M-mode.
- c. In case `mseccfg.MMWP` is not set, M-mode can still access and execute any region not covered by a PMP rule. Since we try to prevent M-mode from executing malicious code and since an attacker may manage to place code on some region not covered by PMP (e.g. a directly-addressable flash memory), we need to ensure that M-mode can only execute the code segments initialized during firmware / OS initialization.
- d. We are only using the encoding `pmppcfg.RW=01` together with `mseccfg.MML`, if `mseccfg.MML` is not set the encoding remains usable for future use.

Part III: The RISC-V Instruction Set Manual, Volume III: Profiles

DRAFT

Preface

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

Profile
RVI20U32
RVI20U64
RVA20U64
RVA20S64
RVA22U64
RVA22S64
RVA23U64
RVA23S64
RVB23U64
RVB23S64

Changes made since ratification of RVA22 profiles

- Clarified that Zihpm was optional in RVA20U64 and became mandatory in RVA22U64

Changes made since public review of RVA22 profiles

- Clarified that profile name can be used as ISA base string
- Renamed Ssptead to Svade
- Fixed Ssu64xl to make supporting UXL=64 mandatory
- Added section listing new extension names in profiles document
- Added new extension name Sscounterenw
- Removed outdated text on Zicntr/Zihpm ratification plan

Chapter 1. Introduction

RISC-V was designed to provide a highly modular and extensible instruction set, and includes a large and growing set of standard extensions. In addition, users may add their own custom extensions. This flexibility can be used to highly optimize a specialized design by including only the exact set of ISA features required for an application, but the same flexibility also leads to a combinatorial explosion in possible ISA choices. Profiles specify a much smaller common set of ISA choices that capture the most value for most users, and which thereby enable the software community to focus resources on building a rich software ecosystem with application and operating system portability across different implementations.



Another pragmatic concern is the long and unwieldy ISA strings required to encode common sets of extensions, which will continue to grow as new extensions are defined.

Each profile is built on a standard base ISA plus a set of mandatory ISA extensions, and provides a small set of standard ISA options to extend the mandatory components. Profiles provide a convenient shorthand for describing the ISA portions of hardware and software platforms, and also guide the development of common software toolchains shared by different platforms that use the same profile. The intent is that the software ecosystem focus on supporting the profiles' mandatory base and standard options, instead of attempting to support every possible combination of individual extensions. Similarly, hardware vendors should aim to structure their offerings around standard profiles to increase the likelihood their designs will have mainstream software support.



Profiles are not intended to prohibit the use of combinations of individual ISA extensions or the addition of custom extensions, which can continue to be used for more specialized applications albeit without the expectation of widespread software support or portability between hardware platforms.



As RISC-V evolves over time, the set of ISA features will grow, and new platforms will be added that may need different profiles. To manage this evolution, RISC-V is adopting a model of regular annual releases of new ISA profiles, following an ISA roadmap managed by the RISC-V Technical Steering Committee. The architecture profiles will also be used for branding and to advertise compatibility with the RISC-V standard.

This volume describes the general structure of RISC-V architecture profiles and also the specifics of the officially defined profiles.

1.1. Profiles versus Platforms

Profiles only describe ISA features, not a complete execution environment.

A *software platform* is a specification for an execution environment, in which software targeted for that software platform can run.

A *hardware platform* is a specification for a hardware system (which can be viewed as a physical realization of an execution environment).

Both software and hardware platforms include specifications for many features beyond details of the ISA used by RISC-V in the platform (e.g., boot process, calling convention, behavior of environment calls, discovery mechanism, presence of certain memory-mapped hardware devices, etc.). Architecture profiles factor out ISA-specific definitions from platform definitions to allow ISA profiles to be reused across different platforms, and to be used by tools (e.g., compilers) that are common across many different platforms.

A platform can add additional constraints on top of those in a profile. For example, mandating an

extension that is a standard option in the underlying profile, or constraining some implementation-specific parameter in the profile to lie within a certain range.

A platform cannot remove mandates or reduce other requirements in a profile.



A new profile should be proposed if existing profiles do not match the needs of a new platform.

1.2. Components of a Profile

1.2.1. Profile Family

Every profile is a member of a *profile family*. A profile family is a set of profiles that share the same base ISA but which vary in highest-supported privilege mode. The profile families defined in this volume are:

- Generic unprivileged instructions (I)
- Application processors running rich operating systems with binary software ecosystems (A)
- Application processors running rich operating systems with rooms for customization (B)



More profile families may be added over time.

A profile family may be updated no more than annually, and the release calendar year is treated as part of the profile family name.

Each profile family is described in more detail below.

1.2.2. Profile Privilege Mode

RISC-V has a layered architecture supporting multiple privilege modes, and most RISC-V platforms support more than one privilege mode. Software is usually written assuming a particular privilege mode during execution. For example, application code is written assuming it will be run in user mode, and kernel code is written assuming it will be run in supervisor mode.



Software can be run in a mode different than the one for which it was written. For example, privileged code using privileged ISA features can be run in a user-mode execution environment, but will then cause traps into the enclosing execution environment when privileged instructions are executed. This behavior might be exploited, for example, to emulate a privileged execution environment using a user-mode execution environment.

The profile for a privilege mode describes the ISA features for an execution environment that has the eponymous privilege mode as the most-privileged mode available, but also includes all supported lower-privilege modes. In general, available instructions vary by privilege mode, and the behavior of RISC-V instructions can depend on the current privilege mode. For example, an **S-mode** profile includes U-mode as well as S-mode and describes the behavior of instructions when running in different modes in an S-mode execution environment, such as how an **ecall** instruction in U-mode causes a contained trap into an S-mode handler whereas an **ecall** in S-mode causes a requested trap out to the execution environment.

A profile may specify that certain conditions will cause a requested trap (such as an **ecall** made in the highest-supported privilege mode) or fatal trap to the enclosing execution environment. The profile does not specify the behavior of the enclosing execution environment in handling requested or fatal traps.



In particular, a profile does not specify the set of ECALLs available in the outer execution environment. This should be documented in the appropriate binary interface to the outer

execution environment (e.g., Linux user ABI, or RISC-V SEE).



In general, a profile can be implemented by an execution environment using any hardware or software technique that provides compatible functionality, including pure software emulation.

A profile does not specify any invisible traps.



In particular, a profile does not constrain how invisible traps to a more-privileged mode can be used to emulate profile features.

A more-privileged profile can always support running software to implement a less-privileged profile from the same profile family. For example, a platform supporting the S-mode profile can run a supervisor-mode operating system that provides user-mode execution environments supporting the U-mode profile.



Instructions in a U-mode profile, which are all executed in user mode, have potentially different behaviors than instructions executed in user mode in an S-mode profile. For this reason, a U-mode profile cannot be considered a subset of an S-mode profile.

1.2.3. Profile ISA Features

An architecture profile has a mandatory ratified base instruction set (RV32I or RV64I for the current profiles). The profile also includes ratified ISA extensions placed into two categories:

1. Mandatory
2. Optional

As the name implies, *Mandatory ISA extensions* are a required part of the profile. Implementations of the profile must provide these. The combination of the profile base ISA plus the mandatory ISA extensions are termed the profile *mandates*, and software using the profile can assume these always exist.

The *Optional* category (also known as *options*) contains extensions that may be added as options, and which are expected to be generally supported as options by the software ecosystem for this profile.



The level of "support" for an Optional extension will likely vary greatly among different software components supporting a profile. Users would expect that software claiming compatibility with a profile would make use of any available supported options, but as a bare minimum software should not report errors or warnings when supported options are present in a system.

An optional extension may comprise many individually named and ratified extensions but a profile option requires all constituent extensions are present. In particular, unless explicitly listed as a profile option, individual extensions are not by themselves a profile option even when required as part of a profile option. For example, the Zbkb extension is not by itself a profile option even though it is a required component of the Zkn option.



Profile optional extensions are intended to capture the granularity at which the broad software ecosystem is expected to cope with combinations of extensions.

All components of a ratified profile must themselves have been ratified.

Platforms may provide a discovery mechanism to determine what optional extensions are present.

Extensions that are not explicitly listed in the mandatory or optional categories are termed *non-profile* extensions, and are not considered parts of the profile. Some non-profile extensions can be added to an implementation without conflicting with the mandatory or optional components of a profile. In this case, the implementation is still compatible with the profile even though additional non-profile extensions are

present. Other non-profile extensions added to an implementation might alter or conflict with the behavior of the mandatory or optional extensions in a profile, in which case the implementation would not be compatible with the profile.



Extensions that are released after a given profile is released are by definition non-profile extensions. For example, mandatory or optional profile extensions for a new profile might be prototyped as non-profile extensions on an earlier profile.

1.2.4. Profile Naming Convention

A profile name is a string comprised of, in order:

1. Prefix **RV** for RISC-V.
2. A specific profile family name string. Currently a single letter (**I**, **A**, or **B**), but later profiles may have longer family name strings.
3. A numeric string giving the first complete calendar year for which the profile is ratified, represented as number of years after year 2000, i.e., **20** for profiles built on specifications ratified during 2019. The year string will be longer than two digits in the next century.
4. A privilege mode (**U**, **S**, **M**). Hypervisor support is treated as an option.
5. A base ISA XLEN specifier (**32**, **64**).

The initial profiles based on specifications ratified in 2019 are:

- RVI20U32 basic unprivileged instructions for RV32I
- RVI20U64 basic unprivileged instructions for RV64I
- RVA20U64, RVA20S64 64-bit application-processor profiles



Profile names are embeddable into RISC-V ISA naming strings. This implies that there will be no standard ISA extension with a name that matches the profile naming convention. This allows tools that process the RISC-V ISA naming string to parse and/or process a combined string.

1.2.5. RVA Profiles Rationale

RISC-V was designed to provide a highly modular and extensible instruction set and includes a large and growing set of standard extensions, where each standard extension is a bundle of instruction-set features. This is no different than other industry ISAs that continue to add new ISA features. Unlike other ISAs, however, RISC-V has a broad set of contributors and implementers, and also allows users to add their own custom extensions. For some deep embedded markets, highly customized processor configurations are desirable for efficiency, and all software is compiled, ported, and/or developed in-house by the same organization for that specific processor configuration. However, for other markets that expect a substantial fraction of software to be delivered to end-customers in binary form, compatibility across multiple implementations from different RISC-V vendors is required.

The RVIA ISA extension ratification process ensures that all processor vendors have agreed to the specification of a standard extension if present. However, by themselves, the ISA extension specifications do not guarantee that a certain set of standard extensions will be present in all implementations.

The primary goal of the RVA profiles is to align processor vendors targeting binary software markets, so software can rely on the existence of a certain set of ISA features in a particular generation of RISC-V implementations.

Alignment is not only for compatibility, but also to ensure RISC-V is competitive in these markets. The binary app markets are also generally those with the most competitive performance requirements (e.g., mobile, client, server). RVIA cannot mandate the ISA features that a RISC-V binary software ecosystem should use, as each ecosystem will typically select the lowest-common denominator they empirically observe in the deployed devices in their target markets. But RVIA can align hardware vendors to support a common set of features in each generation through the RVA profiles. Without proactive alignment through RVA profiles, RISC-V will be uncompetitive, as even if a particular vendor implements a certain feature, if other vendors do not, then binary distributions will not generally use that feature and all implementations will suffer. While certain features may be discoverable, and alternate code provided in case of presence/absence of a feature, the added cost to support such options is only justified for certain limited cases, and binary app markets will not support a wide range of optional features, particularly for the nascent RISC-V binary app ecosystems.

To maintain alignment and increase RISC-V competitiveness over time, the mandatory set of extensions must increase over time in successive generations of RVA profile. (RVA profiles may eventually have to deprecate previously mandatory instructions, but that is unlikely in the near future.) Note that the RISC-V ISA will continue to evolve, regardless of whether a given software ecosystem settles on a certain generation of profile as the baseline for their ecosystem for many years or even decades. There are many existing binary software ecosystems, which will migrate to RISC-V and evolve at different rates, and more new ones will doubtless be created over the hopefully long lifetime of RISC-V. High-performance application processors require considerable investment, and no single binary app ecosystem can justify the development costs of these processors, especially for RISC-V in its early stage of adoption.

While the heart of the profile is the set of mandatory extensions, there are several kinds of optional extension that serve important roles in the profile.

The first kind are *localized options*, whose presence or use necessarily differs along geo-political and/or jurisdictional boundaries, with crypto being the obvious example. These will always be optional. At least for crypto, discovery has been found to be perfectly acceptable to handle this optionality on other architectures, as the use of the extensions is well contained in certain libraries.

The second kind of optional extension is a *development option*, which represents a new ISA extension in an early part of its lifecycle but which is intended to become mandatory in a later generation of the RVA profile. Processor vendors and software toolchain providers will have varying development schedules, and providing an optional phase in a new extension's lifecycle provides some flexibility while maintaining overall alignment, and is particularly appropriate when hardware or software development for the extension is complex. Denoting an extension as a *development option* signals to the community that development should be prioritized for such extensions as they will become mandatory.

The third kind of optional extension are *expansion options*, which are those that may have a large implementation cost but are not always needed in a particular platform, and which can be readily handled by discovery. These are also intended to remain available as expansion options in future versions of the profile. Several supervisor-mode extensions fall into this category, e.g., Sv57, which has a notable PPA impact over Sv48 and is not needed on smaller platforms. Some unprivileged extensions that may fall into this category are possible future matrix extensions. These have large implementation costs, and use of matrix instructions can be readily supported with discovery and alternate math libraries.

The fourth kind of optional extensions are *transitory options*, where it is not clear if the extension will change to a mandatory, localized, or expansion option, or be possibly dropped over time. Cryptography provides some examples where earlier cyphers have been broken and are now deprecated. RVIA used this mechanism to enable scalar crypto until vector crypto was ready. Software security features may also be in this category, with examples of deprecated security features occurring in other architectures. As another example, the recent avalanche of new numeric datatypes for AI/ML may eventually subside with a few survivors actually being used longer term. Denoting an option as transitory signals to the community that this extension may be removed in a future profile, though the time scale may span many years.

Except for the localized options, it could be argued that other three kinds of option could be left out of profiles. Binary distributions of applications willing to invest in discovery can use an optional extension, and customers compiling their own applications can take advantage of the feature on a particular implementation, even when that system is mostly running binary distributions that ignore the new extension. However, there is value in providing guidance to align hardware vendors and software developers around what extensions are worth implementing and worth discovering, by designating only a few important features as profile options and limiting their granularity.

DRAFT

Chapter 2. RVI20 Profiles

The RVI20 profiles document the initial set of unprivileged instructions. These provide a generic target for software toolchains and represent the minimum level of compatibility with RISC-V ratified standards. The two profiles RVI20U32 and RVI20U64 correspond to the RV32I and RV64I base ISAs respectively.



These are designed as unprivileged profiles as opposed to user-mode profiles. Code using this profile can run in any privilege mode, and so requested and fatal traps may be horizontal traps into an execution environment running in the same privilege mode.

2.1. RVI20U32

RVI20U32 specifies the ISA features available to generic unprivileged execution environments.

2.1.1. RVI20U32 Mandatory Base

RV32I is the mandatory base ISA for RVI20U32, and is little-endian.

As per the unprivileged architecture specification, the **ecall** instruction causes a requested trap to the execution environment.

Misaligned loads and stores might not be supported.

The **fence.tso** instruction is mandatory.



*The **fence.tso** instruction was incorrectly described as optional in the 2019 ratified specifications. However, **fence.tso** is encoded within the standard **fence** encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that **fence.tso** is being exploited by a hardware implementation, there is no advantage to making the instruction an option. Later versions of the unprivileged ISA specifications correctly indicate that **fence.tso** is mandatory.*

2.1.2. RVI20U32 Mandatory Extensions

There are no mandatory extensions for RVI20U32.

2.1.3. RVI20U32 Optional Extensions

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.



*The rationale to not include **Q** as an optional extension is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect software to expend effort optimizing use of **Q** instructions in case they are present.*

- **C** Compressed Instructions.
- **Zifencei** Instruction-fetch fence instruction.

- Misaligned loads and stores may be supported.
- **Zicntr** Basic counters.



The Zicsr extension is not supported independent of the Zicntr or F extensions.

- **Zihpm** Hardware performance counters.

2.2. RVI20U64

RVI20U64 specifies the ISA features available to generic unprivileged execution environments.

2.2.1. RVI20U64 Mandatory Base

RV64I is the mandatory base ISA for RVI20U64, and is little-endian.

As per the unprivileged architecture specification, the **ecall** instruction causes a requested trap to the execution environment.

Misaligned loads and stores might not be supported.

The **fence.tso** instruction is mandatory.



*The **fence.tso** instruction was incorrectly described as optional in the 2019 ratified specifications. However, **fence.tso** is encoded within the standard **fence** encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that **fence.tso** is being exploited by a hardware implementation, there is no advantage to making the instruction a profile option. Later versions of the unprivileged ISA specifications correctly indicate that **fence.tso** is mandatory.*

2.2.2. RVI20U64 Mandatory Extensions

There are no mandatory extensions for RVI20U64.

2.2.3. RVI20U64 Optional Extensions

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.



The rationale to not include Q as a profile option is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect software to expend effort optimizing use of Q instructions in case they are present.

- **C** Compressed Instructions.
- **Zifencei** Instruction-fetch fence instruction.
- Misaligned loads and stores may be supported.
- **Zicntr** Basic counters.



The Zicsr extension is not supported independent of the Zicntr or F extensions.

- [Zihpm](#) Hardware performance counters.

DRAFT

Chapter 3. RVA20 Profiles

The RVA20 profiles are intended to be used for 64-bit application processors running rich OS stacks. Only user-mode (RVA20U64) and supervisor-mode (RVA20S64) profiles are specified in this family.



There is no machine-mode profile currently defined for application processor families. A machine-mode profile for application processors would only be used in specifying platforms for portable machine-mode software. Given the relatively low volume of portable M-mode software in this domain, the wide variety of potential M-mode code, and the very specific needs of each type of M-mode software, we are not specifying individual M-mode ISA requirements in the A-family profiles.



Only XLEN=64 application processor profiles are currently defined. It would be possible to also define very similar XLEN=32 variants.

3.1. RVA20U64 Profile

The RVA20U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within the application processor family in terms of the amount of software that targets this profile.

RVA20U64 has one optional extension (Zihpm).

3.1.1. RVA20U64 Mandatory Base

RV64I is the mandatory base ISA for RVA20U64, and is little-endian.

As per the unprivileged architecture specification, the `ecall` instruction causes a requested trap to the execution environment.

The `fence.tso` instruction is mandatory.



The `fence.tso` instruction was incorrectly described as optional in the 2019 ratified specifications. However, `fence.tso` is encoded within the standard `fence` encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that `fence.tso` is being exploited by a hardware implementation, there is no advantage to making the instruction a profile option. Later versions of the unprivileged ISA specifications correctly indicate that `fence.tso` is mandatory.

3.1.2. RVA20U64 Mandatory Extensions

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.
- **C** Compressed Instructions.
- **Zicsr** CSR instructions. These are implied by presence of Zicntr or F.
- **Zicntr** Basic counters.

- **Ziccif** Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to $\min(\text{ILEN}, \text{XLEN})$ (i.e., 32 bits for RVA20) are atomic.



Ziccif is a new extension name capturing this feature. The fetch atomicity requirement facilitates runtime patching of aligned instructions.

- **Ziccrse** Main memory regions with both the cacheability and coherence PMAs must support RsrvEventual.



Ziccrse is a new extension name capturing this feature.

- **Ziccamosa** Main memory regions with both the cacheability and coherence PMAs must support AMOArithmetic.



Ziccamosa is a new extension name capturing this feature.

- **Za128rs** Reservation sets must be contiguous, naturally aligned, and at most 128 bytes in size.



Za128rs is a new extension name capturing this feature. The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

- **Zicclsm** Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.



This introduces a new extension name for this feature. This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

3.1.3. RVA20U64 Optional Extensions

- **Zihpm** Hardware performance counters.



Hardware performance counters are a supported option in RVA20. The number of counters is platform-specific.



The rationale to not make Q an optional extension is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect A-profile software to expend effort optimizing use of Q instructions in case they are present.



*Zifencei is not classed as a supported option in the user-mode profile because it is not sufficient by itself to produce the desired effect in a multiprogrammed multiprocessor environment without OS support, and so the instruction cache flush should always be performed using an OS call rather than using the **fence.i** instruction. **fence.i** semantics can be expensive to implement for some hardware memory hierarchy designs, and so alternative non-standard instruction-cache coherence mechanisms can be used behind the OS abstraction. A separate extension is being developed for more general and efficient instruction cache coherence.*



The execution environment must provide a means to synchronize writes to instruction memory with instruction fetches, the implementation of which likely relies on the Zifencei extension. For example, RISC-V Linux supplies the `__riscv_flush_icache` system call and a corresponding vDSO call.

3.1.4. RVA20U64 Recommendations

Recommendations are not strictly mandated but are included to guide implementers making design choices.

Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

3.2. RVA20S64 Profile

The RVA20S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA20S64 is based on privileged architecture version 1.11.

RVA20S64 has one unprivileged option (Zihpm) and one privileged option (Sv48).

3.2.1. RVA20S64 Mandatory Base

RV64I is the mandatory base ISA for RVA20S64, and is little-endian.

The **ecall** instruction operates as per the unprivileged architecture specification. An **ecall** in user mode causes a contained trap to supervisor mode. An **ecall** in supervisor mode causes a requested trap to the execution environment.

3.2.2. RVA20S64 Mandatory Extensions

The following unprivileged extensions are mandatory:

- The RVA20S64 mandatory unprivileged extensions include all the mandatory unprivileged extensions in RVA20U64.
- **Zifencei** Instruction-Fetch Fence.



Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA20 application processors. A new instruction-cache coherence mechanism is under development which might be added as an option in the future.

The following privileged extensions are mandatory:

- **Ss1p11** Privileged Architecture version 1.11.
- **Svbare** The **satp** mode Bare must be supported.



This is a new extension name for this feature.

- **Sv39** Page-Based 39-bit Virtual-Memory System.
- **Svade** Page-fault exceptions are raised when a page is accessed when A bit is clear, or written when D bit is clear.



This is a new extension name for this feature.

- **Ssccptr** Main memory regions with both the cacheability and coherence PMAs must support hardware page-table reads.



This is a new extension name for this feature.

- **Sstvecd** `stvec.MODE` must be capable of holding the value 0 (Direct). When `stvec.MODE=Direct`, `stvec.BASE` must be capable of holding any valid four-byte-aligned address.



This is a new extension name for this feature.

- **Sstvala** `stval` must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions that are defined to write an address to `stval`, other than those caused by execution of the `EBREAK` or `C.EBREAK` instructions. For virtual-instruction and illegal-instruction exceptions, `stval` must be written with the faulting instruction.



This is a new extension name for this feature.

3.2.3. RVA20S64 Optional Extensions

RVA20S64 has one unprivileged option.

- **Zihpm** Hardware performance counters.



The number of counters is platform-specific.

RVA20S64 has the following privileged options:

- **Sv48** Page-Based 48-bit Virtual-Memory System.
- **Ssu64xl** `sstatus.UXL` must be capable of holding the value 2 (i.e., `UXLEN=64` must be supported).



This is a new extension name for this feature.

Chapter 4. RVA22 Profiles

The RVA22 profiles are intended to be used for 64-bit application processors running rich OS stacks. Only user-mode (RVA22U64) and supervisor-mode (RVA22S64) profiles are specified in this family.

4.1. RVA22U64 Profile

The RVA22U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within the application processor family in terms of the amount of software that targets this profile.

4.1.1. RVA22U64 Mandatory Base

RV64I is the mandatory base ISA for RVA22U64, including mandatory **fence.tso**, and is little-endian.



*Later versions of the RV64I unprivileged ISA specification ratified in 2021 made clear that **fence.tso** is mandatory.*

As per the unprivileged architecture specification, the **ecall** instruction causes a requested trap to the execution environment.

4.1.2. RVA22U64 Mandatory Extensions

The following mandatory extensions were present in RVA20U64.

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.
- **C** Compressed Instructions.
- **Zicsr** CSR instructions. These are implied by presence of F.
- **Zicntr** Base counters and timers.
- **Ziccif** Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to min(ILEN,XLEN) (i.e., 32 bits for RVA22) are atomic.
- **Ziccrse** Main memory regions with both the cacheability and coherence PMAs must support RsrEventual.



Ziccrse is a new extension name capturing this feature.

- **Ziccamao** Main memory regions with both the cacheability and coherence PMAs must support AMOArithmetic.



Ziccamao is a new extension name capturing this feature.

- **Zicclsm** Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.



This is a new extension name for this feature. Even though mandated, misaligned loads and

stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

The following mandatory feature was further restricted in RVA22U64:

- **Za64rs** Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes.



This is a new extension name capturing this feature. The maximum reservation size has been reduced to match the required cache block size. The minimum reservation size is effectively set by the instructions in the mandatory A extension.

The following mandatory extensions are new for RVA22U64.

- **B** Bit-manipulation instructions.



The B extension comprises the Zba, Zbb, and Zbs extensions. At the time of RVA22U64's ratification, the B extension had not yet been defined, and so RVA22U64 explicitly mandated Zba, Zbb, and Zbs instead. Mandating B is equivalent.

- **Zihpm** Hardware performance counters.



Zihpm was optional in RVA20U64.

- **Zihintpause** Pause instruction.



*While the **pause** instruction is a HINT can be implemented as a NOP and hence trivially supported by hardware implementers, its inclusion in the mandatory extension list signifies that software should use the instruction whenever it would make sense and that implementors are expected to exploit this information to optimize hardware execution.*

- **Zic64b** Cache blocks must be 64 bytes in size, naturally aligned in the address space.



This is a new extension name for this feature. While the general RISC-V specifications are agnostic to cache block size, selecting a common cache block size simplifies the specification and use of the following cache-block extensions within the application processor profile. Software does not have to query a discovery mechanism and/or provide dynamic dispatch to the appropriate code. We choose 64 bytes at it is effectively an industry standard. Implementations may use longer cache blocks to reduce tag cost provided they use 64-byte sub-blocks to remain compatible. Implementations may use shorter cache blocks provided they sequence cache operations across the multiple cache blocks comprising a 64-byte block to remain compatible.

- **Zicbom** Cache-Block Management Operations.

- **Zicbop** Cache-Block Prefetch Operations.



As with other HINTS, the inclusion of prefetches in the mandatory set of extensions indicates that software should generate these instructions where they are expected to be useful, and hardware is expected to exploit that information.

- **Zicboz** Cache-Block Zero Operations.

- **Zfhmin** Half-Precision Floating-point transfer and convert.



Zfhmin is a small extension that adds support to load/store and convert IEEE 754 half-precision numbers to and from the IEEE 754 single-precision format. The hardware cost for this extension is low, and mandating the extension avoids adding an option to the profile.

- **Zkt** Data-independent execution time.



Zkt requires a certain subset of integer instructions execute with data-independent latency. Mandating this feature enables portable libraries for safe basic cryptographic operations. It is expected that application processors will naturally have this property and so implementation cost is low, if not zero, in most systems that would support RVA22.

4.1.3. RVA22U64 Optional Extensions

RVA22U64 has four profile options (Zfh, V, Zkn, Zks):

- **Zfh** Half-Precision Floating-Point.



A future profile might mandate Zfh.

- **V** Vector Extension.



The smaller vector extensions (Zve32f, Zve32x, Zve64d, Zve64f, Zve64x) are not provided as separately supported profile options. The full V extension is specified as the only supported profile option.



A future profile might mandate V.

- **Zkn** Scalar Crypto NIST Algorithms.
- **Zks** Scalar Crypto ShangMi Algorithms.



The scalar crypto extensions are expected to be superseded by vector crypto standards in future profiles, and the scalar extensions may be removed as supported options once vector crypto is present.



The smaller component scalar crypto extensions (Zbc, Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh) are not provided as separate options in the profile. Profile implementers should provide all of the instructions in a given algorithm suite as part of the Zkn or Zks supported options.



Access to the entropy source (Zkr) in a system is usually carefully controlled. While the design supports unprivileged access to the entropy source, this is unlikely to be commonly used in an application processor, and so Zkr was not added as a profile option. This also means the roll-up Zk was not added as a profile option.



The Zfinx, Zdinx, Zhinx, Zhinxmin extensions are incompatible with the profile mandates to support the F and D extensions.

4.1.4. RVA22U64 Recommendations

Recommendations are not strictly mandated but are included to guide implementers making design choices.

Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

4.2. RVA22S64 Profile

The RVA22S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA22S64 is based on privileged architecture version 1.12.

4.2.1. RVA22S64 Mandatory Base

RV64I is the mandatory base ISA for RVA22S64, including mandatory **fence.tso**, and is little-endian.



*Later versions of the RV64I unprivileged ISA specification ratified in 2021 made clear that **fence.tso** is mandatory.*

The **ecall** instruction operates as per the unprivileged architecture specification. An **ecall** in user mode causes a contained trap to supervisor mode. An **ecall** in supervisor mode causes a requested trap to the execution environment.

4.2.2. RVA22S64 Mandatory Extensions

The following unprivileged extensions are mandatory:

- The RVA22S64 mandatory unprivileged extensions include all the mandatory unprivileged extensions in RVA22U64.

- **Zifencei** Instruction-Fetch Fence.



Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA22 application processors. A new instruction-cache coherence mechanism is under development which might be added as an option in the future.

The following privileged extensions are mandatory:

- **Ss1p12** Privileged Architecture version 1.12.



Ss1p12 supersedes Ss1p11.

- **Svbare** The **satp** mode Bare must be supported.



This is a new extension name for this feature.

- **Sv39** Page-Based 39-bit Virtual-Memory System.
- **Svade** Page-fault exceptions are raised when a page is accessed when A bit is clear, or written when D bit is clear.
- **Ssccptr** Main memory regions with both the cacheability and coherence PMAs must support hardware page-table reads.
- **Sstvecd** **stvec.MODE** must be capable of holding the value 0 (Direct). When **stvec.MODE=Direct**, **stvec.BASE** must be capable of holding any valid four-byte-aligned address.
- **Sstvala** **stval** must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions other than those caused by execution of the EBREAK or C.EBREAK instructions. For virtual-instruction and illegal-instruction exceptions, **stval** must be written with the faulting instruction.
- **Sscounterenw** For any hpmcounter that is not read-only zero, the corresponding bit in **scounterenw** must be writable.



This is new extension name capturing this feature.

- **Svpbmt** Page-Based Memory Types
- **Svinval** Fine-Grained Address-Translation Cache Invalidation

4.2.3. RVA22S64 Optional Extensions

RVA22S64 has four unprivileged options (Zfh, V, Zkn, Zks) from RVA22U64, and eight privileged options (Sv48, Sv57, Svnapot, Ssu64xl, Sstc, Sscofpmf, Zkr, H).

The privileged optional extensions are:

- **Sv48** Page-Based 48-bit Virtual-Memory System.
- **Sv57** Page-Based 57-bit Virtual-Memory System.
- **Svnapot** NAPOT Translation Contiguity



It is expected that Svnapot will be mandatory in the next profile release.

- **Ssu64xl** **ssstatus.UXL** must be capable of holding the value 2 (i.e., UXLEN=64 must be supported).



This is a new extension name for this feature.

- **Sstc** supervisor-mode timer interrupts.



Sstc was not made mandatory in RVA22S64 as it is a more disruptive change affecting system-level architecture, and will take longer for implementations to adopt. It is expected to be made mandatory in the next profile release.

- **Sscofpmf** Count Overflow and Mode-Based Filtering.



Platforms may choose to mandate the presence of Sscofpmf.

- **Zkr** Entropy CSR.



Technically, Zk is also a privileged-mode option capturing that Zkr, Zkn, and Zkt are all implemented. However, the Zk rollup is less descriptive than specifying the individual extensions explicitly.

- **Sha** The augmented hypervisor extension. Sha comprises the following extensions:
 - **H** The hypervisor extension.
 - **Ssstateen** Supervisor-mode view of the state-enable extension. The supervisor-mode (**ssstateen0-3**) and hypervisor-mode (**hstateen0-3**) state-enable registers must be provided.
 - **Shcounterenw** For any **hpmcounter** that is not read-only zero, the corresponding bit in **hcounteren** must be writable.
 - **Shvstvala** **vstval** must be written in all cases described above for **stval**.
 - **Shtvala** **htval** must be written with the faulting guest physical address in all circumstances permitted by the ISA.
 - **Shvstvecd** **vstvec.MODE** must be capable of holding the value 0 (Direct). When **vstvec.MODE**=Direct, **vstvec.BASE** must be capable of holding any valid four-byte-aligned address.
 - **Shvsatpa** All translation modes supported in **satp** must be supported in **vsatp**.

- **Shgatpa** For each supported virtual memory scheme SvNN supported in **satp**, the corresponding h gatp SvNNx4 mode must be supported. The h gatp mode Bare must also be supported.



Sha, Shcounterenw, Shvstvala, Shtvala, Shvstvecd, Shvsatpa, and Shgatpa are new extension names. Sha was introduced after RVA22S64 was ratified, but the ratified text required all of Sha's constituent extensions if the optional H extension was included. Hence, offering the Sha option is equivalent.



The Smstateen extension specification is an M-mode extension as it includes M-mode features, but the supervisor-mode visible components of the extension are named as the Ssstateen extension. Only Ssstateen is mandated in the RVA22S64 profile when the hypervisor extension is implemented. These registers are not mandated or supported options without the hypervisor extension, as there are no RVA22S64 supported options with relevant state to control in the absence of the hypervisor extension.

4.2.4. RVA22S64 Recommendations

- Implementations are strongly recommended to raise illegal-instruction exceptions when attempting to execute unimplemented opcodes.

DRAFT

Chapter 5. RVA23 Profiles

The RVA23 profiles are intended to align implementations of RISC-V 64-bit application processors to allow binary software ecosystems to rely on a large set of guaranteed extensions and a small number of discoverable coarse-grain options. It is explicitly a non-goal of RVA23 to allow more hardware implementation flexibility by supporting only a minimal set of features and a large number of fine-grain extensions.

Only user-mode (RVA23U64) and supervisor-mode (RVA23S64) profiles are specified in this family.

5.1. RVA23U64 Profile

The RVA23U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within the application processor family in terms of the amount of software that targets this profile.

5.1.1. RVA23U64 Mandatory Base

RV64I is the mandatory base ISA for RVA23U64 and is little-endian. As per the unprivileged architecture specification, the **ECALL** instruction causes a requested trap to the execution environment.

5.1.2. RVA23U64 Mandatory Extensions

The following mandatory extensions were present in RVA22U64.

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.
- **C** Compressed instructions.
- **B** Bit-manipulation instructions.
- **Zicsr** CSR instructions. These are implied by presence of F.
- **Zicntr** Base counters and timers.
- **Zihpm** Hardware performance counters.
- **Ziccif** Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to $\min(\text{ILEN}, \text{XLEN})$ (i.e., 32 bits for RVA23) are atomic.
- **Ziccrse** Main memory regions with both the cacheability and coherence PMAs must support RsrsvEventual.
- **Ziccamosa** Main memory regions with both the cacheability and coherence PMAs must support all atomics in the Zaamosa extension.
- **Zicclsm** Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.
- **Za64rs** Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes.
- **Zihintpause** Pause hint.

- **Zic64b** Cache blocks must be 64 bytes in size, naturally aligned in the address space.
- **Zicbom** Cache-block management instructions.
- **Zicbop** Cache-block prefetch instructions.
- **Zicboz** Cache-Block Zero Instructions.
- **Zfhmin** Half-precision floating-point.
- **Zkt** Data-independent execution latency.

The following mandatory extensions are new in RVA23U64:

- **V** Vector extension.



V was optional in RVA22U64.

- **Zvfhmin** Vector minimal half-precision floating-point.
- **Zvbb** Vector basic bit-manipulation instructions.
- **Zvkt** Vector data-independent execution latency.
- **Zihintntl** Non-temporal locality hints.
- **Zicond** Integer conditional operations.
- **Zimop** may-be-operations.
- **Zcmop** Compressed may-be-operations.
- **Zcb** Additional compressed instructions.
- **Zfa** Additional floating-Point instructions.
- **Zawrs** Wait-on-reservation-set instructions.
- **Supm** Pointer masking, with the execution environment providing a means to select PMLen=0 and PMLen=7 at minimum.

5.1.3. RVA23U64 Optional Extensions

5.1.3.1. Localized Options

The following localized options are new in RVA23U64:

- **Zvkng** Vector crypto NIST algorithms with GCM.
- **Zvksg** Vector crypto ShangMi algorithms with GCM.



The scalar crypto extensions Zkn and Zks that were options in RVA22 are not options in RVA23. The goal is for both hardware and software vendors to move to use vector crypto, as vectors are now mandatory and vector crypto is substantially faster than scalar crypto.



We have included only the Zvkng/Zvksg options with GCM to standardize on a higher performance crypto alternative. Zvbc is listed as a development option for use in other algorithms, and will become mandatory. Scalar Zbc is now listed as an expansion option, i.e., it will probably not become mandatory.

5.1.3.2. Development Options

The following are new development options intended to become mandatory in a future RVA profile.

- **Zabha** Byte and halfword atomic memory operations.
- **Zacas** Compare-and-Swap instructions.
- **Ziccamoc** Main memory regions with both the cacheability and coherence PMAs must provide AMOCASQ level PMA support.



Ziccamoc is a new profile-defined extension that ensures Compare and Swap instructions are properly supported in main memory regions. The extension will be added to the PMA section of the privileged architecture manual.

- **Zvbc** Vector carryless multiplication.
- **Zama16b** Misaligned loads, stores, and AMOs to main memory regions that do not cross a naturally aligned 16-byte boundary are atomic.



Zama16b is a new profile-defined extension that represents the presence of the new Misaligned Atomicity Granule feature added in Sm1p13. The extension will be added to the PMA section of the privileged architecture manual.

5.1.3.3. Expansion Options

The following expansion options were also present in RVA22U64:

- **Zfh** Scalar half-precision floating-point.

The following are new expansion options in RVA23U64:

- **Zbc** Scalar carryless multiply.
- **Zicfilp** Landing Pads.
- **Zicfiss** Shadow Stack.
- **Zvfh** Vector half-precision floating-point.
- **Zfbfmin** Scalar BF16 converts.
- **Zvfbfmin** Vector BF16 converts.
- **Zvfbfwma** Vector BF16 widening mul-add.

5.1.3.4. Transitory Options

There are no transitory options in RVA23U64.



Scalar crypto is no longer an option in RVA23U64, though the Zbc extension has now been exposed as an expansion option.

5.1.4. RVA23U64 Recommendations

Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

5.2. RVA23S64 Profile

The RVA23S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA23S64 is based on privileged architecture version 1.13.

5.2.1. RVA23S64 Mandatory Base

RV64I is the mandatory base ISA for RVA23S64 and is little-endian. The **ECALL** instruction operates as per the unprivileged architecture specification. An **ECALL** in user mode causes a contained trap to supervisor mode. An **ECALL** in supervisor mode causes a requested trap to the execution environment.

5.2.2. RVA23S64 Mandatory Extensions

The following unprivileged extensions are mandatory:

- The RVA23S64 mandatory unprivileged extensions include all the mandatory unprivileged extensions in RVA23U64.
- **Zifencei** Instruction-Fetch Fence.



Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA23 application processors. A new instruction-cache coherence mechanism is under development (tentatively named Zjid) which might be added as an option in the future.

The following privileged extensions are mandatory:

- **Ss1p13** Supervisor architecture version 1.13.



Ss1p13 supersedes Ss1p12.

The following privileged extensions were also mandatory in RVA22S64:

- **Svbare** The **satp** mode Bare must be supported.
- **Sv39** Page-based 39-bit virtual-Memory system.
- **Svade** Page-fault exceptions are raised when a page is accessed when A bit is clear, or written when D bit is clear.
- **Ssccptr** Main memory regions with both the cacheability and coherence PMAs must support hardware page-table reads.
- **Sstvecd** **stvec.MODE** must be capable of holding the value 0 (Direct). When **stvec.MODE=Direct**, **stvec.BASE** must be capable of holding any valid four-byte-aligned address.
- **Sstvala** **stval** must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions that are defined to write an address to **stval**, other than those caused by execution of the **EBREAK** or **C.EBREAK** instructions. For virtual-instruction and illegal-instruction exceptions, **stval** must be written with the faulting instruction.
- **Sscounterenw** For any **hpmcounter** that is not read-only zero, the corresponding bit in **scounteren** must be writable.
- **Svpbmt** Page-based memory types
- **Svinval** Fine-grained address-translation cache invalidation.

The following are new mandatory extensions:

- **Svnapot** NAPOT translation contiguity.

 *Svnapot was optional in RVA22.*

- **Sstc** supervisor-mode timer interrupts.

 *Sstc was optional in RVA22.*

- **Sscofpmf** count overflow and mode-based filtering.

- **Ssnpm** Pointer masking, with **senvcfg.PMM** and **henvcfg.PMM** supporting, at minimum, settings **PMLen=0** and **PMLen=7**.

- **Ssu64xl sstatus.UXL** must be capable of holding the value 2 (i.e., **UXLen=64** must be supported).

 *Ssu64xl was optional in RVA22.*

- **Sha** The augmented hypervisor extension.



Sha is a new profile-defined extension that captures the full set of features that are mandated to be supported along with the H extension. There is no change to the features added by including the hypervisor extension in a profile—the new name is solely to simplify the text of the profiles. The definition has been added to the RVA22 profile text, where the hypervisor extension was first added, but will be added to the hypervisor section of the combined ISA manual.

Sha comprises the following extensions:

- **H** The hypervisor extension.
- **Ssstateen** Supervisor-mode view of the state-enable extension. The supervisor-mode (**sstateen0-3**) and hypervisor-mode (**hstateen0-3**) state-enable registers must be provided.
- **Shcounterenw** For any **hpmcounter** that is not read-only zero, the corresponding bit in **hcounterenw** must be writable.
- **Shvstvala vstval** must be written in all cases described above for **stval**.
- **Shtvala htval** must be written with the faulting guest physical address in all circumstances permitted by the ISA.
- **Shvstvecd vstvec.MODE** must be capable of holding the value 0 (Direct). When **vstvec.MODE=Direct**, **vstvec.BASE** must be capable of holding any valid four-byte-aligned address.
- **Shvsatpa** All translation modes supported in **satp** must be supported in **vsatp**.
- **Shgatpa** For each supported virtual memory scheme SvNN supported in **satp**, the corresponding **hgatp** SvNNx4 mode must be supported. The **hgatp** mode Bare must also be supported.

 *The augmented hypervisor extension (exactly equivalent to Sha) was optional in RVA22.*

5.2.3. RVA23S64 Optional Extensions

RVA23S64 has the same unprivileged options as RVA23U64.

The privileged options in RVA23S64 are listed in the following sections.

5.2.3.1. Localized Options

There are no privileged localized options in RVA23S64.

5.2.3.2. Development Options

There are no privileged development options in RVA23S64.

5.2.3.3. Expansion Options

The following privileged expansion options were present in RVA22S64:

- **Sv48** Page-based 48-bit virtual-memory system.
- **Sv57** Page-based 57-bit virtual-memory system.
- **Zkr** Entropy CSR.

The following are new privileged expansion options in RVA23S64

- **Svadu** Hardware A/D bit updates.
- **Sdtrig** Debug triggers.
- **Ssstrict** No non-conforming extensions are present. Attempts to execute unimplemented opcodes or access unimplemented CSRs in the standard or reserved encoding spaces raises an illegal instruction exception that results in a contained trap to the supervisor-mode trap handler.



Ssstrict is a new profile-defined extension that restricts the behavior of reserved encoding spaces. The extension will be added to the supervisor chapter of the privileged architecture.



Ssstrict does not prescribe behavior for the custom encoding spaces or CSRs.



Ssstrict definition applies to the execution environment claiming to be RVA23-compatible, which must have the hypervisor extension. That execution environment will take a contained trap to supervisor-mode (however that trap is implemented, including, but not limited to, emulation/delegation in the outer execution environment). Ssstrict (and all the other RVA23 mandates and options) do not apply to any guest VMs run by a hypervisor. An RVA23 hypervisor can provide guest VMs that are also RVA23-compatible but with an expanded set of emulated standard instructions. An RVA23 hypervisor can also choose to implement guest VMs that are not RVA23 compatible (e.g., lacking H, or only RVA20).

- **Svvptc** Transitions from invalid to valid PTEs will be visible in bounded time without an explicit memory-management fence.
- **Sspm** Supervisor-mode pointer masking, with the supervisor execution environment providing a means to select PMLen=0 and PMLen=7 at minimum.

5.2.3.4. Transitory Options

There are no privileged transitory options in RVA23S64.

5.2.4. RVA23S64 Recommendations

- Implementations are strongly recommended to raise illegal-instruction exceptions when attempting to execute unimplemented opcodes or access unimplemented CSRs.

Chapter 6. RVB23 Profiles

6.1. Introduction

This chapter specifies the RVB23 profile family. RVB23 is the first major release of the RVB series of RISC-V Application Processor Profile.

RVB profiles are intended to be used for customized 64-bit application processors that will run rich OS stacks, but usually as a custom build of standard OS source-code distributions. The approach is to provide a large guaranteed set of relatively inexpensive and/or widely beneficial features but allow optionality for more expensive and/or more targeted extensions.

Unlike the RVA profiles, it is explicitly a non-goal of RVB profiles to provide a single standard ISA interface supporting a wide variety of binary kernel and binary application software distributions. However, individual software ecosystems may build upon RVB profiles to produce a more targeted standard interface for a certain market.

6.2. RVB23 Profiles

Only user-mode (RVB23U64) and supervisor-mode (RVB23S64) profiles are specified in this family.

6.2.1. RVB23U64 Profile

The RVB23U64 profile specifies the ISA features available to user-mode execution environments in 64-bit RVB applications processors.

6.2.1.1. RVB23U64 Mandatory Base

RV64I is the mandatory base ISA for RVB23U64 and is little-endian. As per the unprivileged architecture specification, the **ECALL** instruction causes a requested trap to the execution environment.

6.2.1.2. RVB23U64 Mandatory Extensions

The following mandatory extensions in RVB23U64 were also mandatory in RVA22U64.

- **M** Integer multiplication and division.
- **A** Atomic instructions.
- **F** Single-precision floating-point instructions.
- **D** Double-precision floating-point instructions.
- **C** Compressed instructions.
- **B** Bit-manipulation instructions.
- **Zicsr** CSR instructions. These are implied by presence of F.
- **Zicntr** Base counters and timers.
- **Zihpm** Hardware performance counters.
- **Ziccif** Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to $\min(\text{ILEN}, \text{XLEN})$ (i.e., 32 bits for RVB23) are atomic.

- **Ziccrse** Main memory regions with both the cacheability and coherence PMAs must support RsrEventual.
- **Ziccamo** Main memory regions with both the cacheability and coherence PMAs must support all atomics in the Zaamo extension.
- **Zicclsm** Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.
- **Za64rs** Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes.
- **Zihintpause** Pause hint.
- **Zic64b** Cache blocks must be 64 bytes in size, naturally aligned in the address space.
- **Zicbom** Cache-block management instructions.
- **Zicbop** Cache-block prefetch instructions.
- **Zicboz** Cache-block zero instructions.
- **Zkt** Data-independent execution latency.

The following mandatory extensions are also present in RVA23U64:

- **Zihintntl** Non-temporal locality hints.
- **Zicond** Integer conditional operations.
- **Zimop** May-be-operations.
- **Zcmop** Compressed may-be-operations.
- **Zcb** Additional compressed instructions.
- **Zfa** Additional floating-point instructions.
- **Zawrs** Wait-on-reservation-set instructions.

6.2.1.3. RVB23U64 Optional Extensions

RVB23U64 has 18 profile options listed below.

Localized Options

The following extensions are localized options in both RVA23U64 and RVB23U64:

- **Zvkng** Vector crypto NIST Algorithms with GCM.
- **Zvksg** Vector crypto ShangMi Algorithms with GCM.

The following extensions options are localized options in RVB23U64 but are not present in RVA23U64:

- **Zvkg** Vector GCM/GMAC instructions.
- **Zvknc** Vector crypto NIST algorithms with carryless multiply.
- **Zvksc** Vector crypto ShangMi algorithms with carryless multiply.



RVA profiles mandate the higher-performing but more expensive GHASH options when adding vector crypto. To reduce implementation cost, RVB profiles also allow these carryless multiply options (Zvknc and Zvksc) to implement GCM efficiently, with GHASH available as a separate option.

- **Zkn** Scalar crypto NIST algorithms.
- **Zks** Scalar crypto ShangMi algorithms.



RVA23 profiles drop support for scalar crypto as an option, as the vector extension is now mandatory in RVA23. RVB23 profiles support scalar crypto, as the vector extension is optional in RVB23.

Development Options

The following are new development options intended to become mandatory in a later RVB profile:

- **Zabha** Byte and halfword atomic memory operations.
- **Zacas** Compare-and-Swap instructions.
- **Ziccamoc** Main memory regions with both the cacheability and coherence PMAs must provide AMOCASQ level PMA support.
- **Zama16b** Misaligned loads, stores, and AMOs to main memory regions that do not cross a naturally aligned 16-byte boundary are atomic.

Expansion Options

The following are expansion options in RVB23U64, but are mandatory in RVA23U64.

- **Zfhmin** Half-precision floating-point.
- **V** Vector extension.



Unclear if other Zve extensions should also be supported in RVB.*

- **Zvfhmin** Vector minimal half-precision floating-point.
- **Zvbb** Vector basic bit-manipulation instructions.
- **Zvkt** Vector data-independent execution latency.
- **Supm** Pointer masking, with the execution environment providing a means to select PMLen=0 and PMLen=7 at minimum.

The following extensions are expansion options in both RVA23U64 and RVB23U64:

- **Zfh** Scalar half-precision floating-point.
- **Zbc** Scalar carryless multiplication.
- **Zicfilp** Landing Pads.
- **Zicfiss** Shadow Stack.
- **Zvfh** Vector half-precision floating-point.
- **Zfbfmin** Scalar BF16 converts.
- **Zvfbfmin** Vector BF16 converts.
- **Zvfbfwma** Vector BF16 widening mul-add.

The following are expansion options for RVB23U64 as they are not intended to be made mandatory in future RVB profiles, but are listed as RVA23U64 development options as they are intended to become mandatory in future RVA profiles.

- **Zvbc** Vector carryless multiplication.

Transitory Options

There are no transitory options in RVB23U64.

6.2.1.4. RVB23U64 Recommendations

Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

6.2.2. RVB23S64 Profile

The RVB23S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVB23S64 is based on privileged architecture version 1.13.



Priv 1.13 is still being defined.

6.2.2.1. RVB23S64 Mandatory Base

RV64I is the mandatory base ISA for RVB23S64 and is little-endian. The **ECALL** instruction operates as per the unprivileged architecture specification. An **ECALL** in user mode causes a contained trap to supervisor mode. An **ECALL** in supervisor mode causes a requested trap to the execution environment.

6.2.2.2. RVB23S64 Mandatory Extensions

The following unprivileged extensions are mandatory:

- The RVB23S64 mandatory unprivileged extensions include all the mandatory unprivileged extensions in RVB23U64.
- **Zifencei** Instruction-Fetch Fence.



Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVB23 application processors. A new instruction-cache coherence mechanism is under development (tentatively named Zjid) which might be added as an option in the future.

The following privileged extensions are mandatory, and are also mandatory in RVA23S64.

- **Ss1p13** Supervisor architecture version 1.13.
- **Ss1p13** *Ss1p13 supersedes Ss1p12 but is not yet ratified.*
- **Svnapot** NAPOT translation contiguity.
- **Svnapot** *Svnapot is very low cost to provide, so is made mandatory even in RVB.*
- **Svbare** The **satp** mode Bare must be supported.
- **Sv39** Page-Based 39-bit Virtual-Memory System.
- **Svade** Page-fault exceptions are raised when a page is accessed when A bit is clear, or written when D bit is clear.
- **Ssccptr** Main memory regions with both the cacheability and coherence PMAs must support hardware

page-table reads.

- **Sstvecd** `stvec.MODE` must be capable of holding the value 0 (Direct). When `stvec.MODE=Direct`, `stvec.BASE` must be capable of holding any valid four-byte-aligned address.
- **Sstvala** `stval` must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for breakpoint exceptions that are defined to write an address to `stval`, other than those caused by execution of the **EBREAK** or **C.EBREAK** instructions. For virtual-instruction and illegal-instruction exceptions, `stval` must be written with the faulting instruction.
- **Sscounterenw** For any `hpmcounter` that is not read-only zero, the corresponding bit in `scounteren` must be writable.
- **Svpbmt** Page-based memory types.
- **Svinval** Fine-grained address-translation cache invalidation.
- **Sstc** supervisor-mode timer interrupts.
- **Sscofpmf** Count overflow and mode-based filtering.
- **Ssu64xl** `sstatus.UXL` must be capable of holding the value 2 (i.e., `UXLEN=64` must be supported).

6.2.2.3. RVB23S64 Optional Extensions

RVB23S64 has the same unprivileged options as RVB23U64.

The privileged options in RVB23S64 are listed in the following sections.

Localized Options

There are no privileged localized options in RVB23S64.

Development Options

There are no privileged development options in RVB23S64.

Expansion Options

The following are privileged expansion options in RVB23S64, but are mandatory in RVA23S64:

- **Ssnpm** Pointer masking, with `senvcfg.PMM` supporting at minimum, settings `PMLen=0` and `PMLen=7`.
- **Sha** The augmented hypervisor extension.

When the hypervisor extension is implemented, the following are also mandatory:

- If the hypervisor extension is implemented and pointer masking (**Ssnpm**) is supported then `henvcfg.PMM` must support at minimum, settings `PMLen=0` and `PMLen=7`.

The following are privileged expansion options in RVB23S64 that are also privileged expansion options in RVA23S64:

- **Sv48** Page-based 48-bit virtual-memory system.
- **Sv57** Page-based 57-bit virtual-memory system.
- **Svadu** Hardware A/D bit updates.

- **Zkr** Entropy CSR.
- **Sdtrig** Debug triggers.
- **Ssstrict** No non-conforming extensions are present. Attempts to execute unimplemented opcodes or access unimplemented CSRs in the standard or reserved encoding spaces raises an illegal instruction exception that results in a contained trap to the supervisor-mode trap handler.



Ssstrict does not prescribe behavior for the custom encoding spaces or CSRs.



Ssstrict definition applies to the execution environment claiming to be RVA23-compatible, which must have the hypervisor extension. That execution environment will take a contained trap to supervisor-mode (however that trap is implemented, including, but not limited to, emulation/delegation in the outer execution environment). Ssstrict (and all the other RVA23 mandates and options) do not apply to any guest VMs run by a hypervisor. An RVA23 hypervisor can provide guest VMs that are also RVA23-compatible but with an expanded set of emulated standard instructions. An RVA23 hypervisor can also choose to implement guest VMs that are not RVA23 compatible (e.g., lacking H, or only RVA20).

- **Svvptc** Transitions from invalid to valid PTEs will be visible in bounded time without an explicit memory-management fence.
- **Sspm** Supervisor-mode pointer masking, with the supervisor execution environment providing a means to select PMLen=0 and PMLen=7 at minimum.

6.2.2.4. RVB23S64 Recommendations

- Implementations are strongly recommended to raise illegal-instruction exceptions when attempting to execute unimplemented opcodes.

Index

@

- (calling convention standard), [156](#)
- (compressed formats), [156](#)
- (compressed. C.BREAKPOINTINSTR), [164](#)
- (compressed. C.CA), [163](#)
- (compressed. C.CR), [163](#)
- (compressed. C.DIINST), [164](#)
- (compressed. C.NOPINSTR), [164](#)
- (register source specifiers c-ext), [156](#)

A

- atomics
 - misaligned, [87](#)

B

- bi-endian, [20](#)

C

- compressed
 - C.ANDI, [163](#)
 - C.SRLI
 - C.SRAI, [162](#)
- cb-format load and store, [161](#)
- CI, [162](#)
- CIW, [162](#)
- cj-format load and store, [160](#)
- cr-format load and store, [160](#)
- cs-format load and store, [159](#)
- integer constant generation, [161](#)
- integer register-immediate, [161](#)
- register-based load and store, [159](#)
- core
 - accelerator, [15](#)
 - cluster
 - multiprocessors, [15](#)
 - component, [15](#)
 - extensions
 - coprocessor, [15](#)

D

- decomposition, [88](#)
- design
 - high performance, [87](#)
 - scalable, [87](#)
- double-precision
 - floating point, [123](#)
 - to single-precision, [125](#)

E

- endian
 - bi-, [20](#)
 - little and big, [20](#)
- exceptions, [21](#)

F

- FENCE, [87](#)
- FENCE.I
 - finer-grained, [47](#)
 - forward compatibility, [47](#)
 - synchronization, [47](#)
- floating point
 - convert and move, [125](#)
 - double precision, [123](#)
 - fused multiply-add, [118](#)
 - load and store, [124](#)
- floating-point
 - classification, [121](#)
 - classify, [126](#)
 - compare, [126](#)
 - conversion, [119](#)
 - exception flag, [116](#)
 - requirements, [123](#)
 - supported precisions, [123](#)

H

- hart
 - execution environment, [16](#)
- HINT
 - PAUSE, [62](#)

I

- ILEN, [20](#)
- IMAFD, [20](#)
- interrupts, [21](#)
- ISA
 - definition, [14](#)

M

- memory access
 - implicit and explicit, [19](#), [19](#)
- MUL
 - DIV, [68](#)
 - div by zero, [69](#)
 - DIVU, [68](#)
 - MULH, [68](#)
 - MULHSU, [68](#)
 - MULHU, [68](#)
 - Zmmul, [69](#)

N

NaN

- generation, [117](#)
- propagation, [117](#)

O

operations

- memory, [87](#)
- subnormal, [117](#)

P

PAUSE

- duration, [62](#)
- encoding, [62](#)
- energy consumption, [62](#)
- HINT, [62](#)
- LR/RC sequences, [62](#)

R

RV32E

- design, [41](#)
- difference from RV32I, [41](#)

RV64I

- compares, [42](#)
- HINT, [44](#)
- LD, [44](#)
- LUI, [43](#)
- RV64I-only, [42](#)
- shifts, [42](#)
- SLLI, [42](#)
- SRAIW, [42](#)
- SRLIW, [42](#)

RV64I-only

- ADDW, [43](#)
- SLLW, [43](#)
- SRAW, [43](#)
- SRLW, [43](#)
- SUBW, [43](#)

RVWMO, [87](#)

S

SFENCE, [87](#)

single-precision

- to double-precision, [125](#)

store instruction word

- not included, [46](#)

T

tininess

- handling, [117](#)

traps, [21](#)

U

unspecified

- behaviors, [22](#)
- values, [22](#)

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/rem-s-project/sail

The RISC-V Debug Specification. github.com/riscv/riscv-debug-spec

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 and analysis. *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.

Goldberg, R. P. (1974). Survey of virtual machine research. *Computer*, 7(6), 34–45.

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., & Verbaauwhede, 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

Moghim, 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/moghim-moghim

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

Navarro, J., Iyer, S., Druschel, P., & Cox, A. (2002). Practical, Transparent Operating System Support for Superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI), 89–104. doi.org/10.1145/844128.844138

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
- Serebryany, K., Stepanov, E., Shlyapnikov, A., Tsyrklevich, V., & Vyukov, D. (2018). Memory Tagging and how it improves C/C++ memory safety. *CoRR*, abs/1802.09517. arxiv.org/abs/1802.09517
- 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[®]