



DRAFT

The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20191214, Revised 20230723

Table of Contents

Preamble	1
Preface	2
1. Introduction	10
1.1. RISC-V Hardware Platform Terminology	11
1.2. RISC-V Software Execution Environments and Harts	11
1.3. RISC-V ISA Overview	12
1.4. Memory	15
1.5. Base Instruction-Length Encoding	16
1.5.1. Expanded Instruction-Length Encoding	16
1.6. Exceptions, Traps, and Interrupts	18
1.7. UNSPECIFIED Behaviors and Values	20
2. RV32I Base Integer Instruction Set, Version 2.1	21
2.1. Programmers' Model for Base Integer ISA	21
2.2. Base Instruction Formats	23
2.3. Immediate Encoding Variants	24
2.4. Integer Computational Instructions	25
2.4.1. Integer Register-Immediate Instructions	26
2.4.2. Integer Register-Register Operations	27
2.4.3. NOP Instruction	28
2.5. Control Transfer Instructions	28
2.5.1. Unconditional Jumps	28
2.5.2. Conditional Branches	30
2.6. Load and Store Instructions	32
2.7. Memory Ordering Instructions	34
2.8. Environment Call and Breakpoints	35
2.9. HINT Instructions	36
3. "Zifencei" Instruction-Fetch Fence, Version 2.0	39
4. "Zihintntnl" Non-Temporal Locality Hints, Version 1.0	41
5. "Zihintpause" Pause Hint, Version 2.0	45
6. RV32E and RV64E Base Integer Instruction Sets, Version 2.0	46
6.1. RV32E and RV64E Programmers' Model	46
6.2. RV32E and RV64E Instruction Set Encoding	46
7. RV64I Base Integer Instruction Set, Version 2.1	47
7.1. Register State	47
7.2. Integer Computational Instructions	47
7.2.1. Integer Register-Immediate Instructions	47
7.2.2. Integer Register-Register Operations	48
7.3. Load and Store Instructions	49
7.4. HINT Instructions	49
8. RV128I Base Integer Instruction Set, Version 1.7	52
9. "M" Standard Extension for Integer Multiplication and Division, Version 2.0	54

9.1. Multiplication Operations	54
9.2. Division Operations	54
9.3. Zmmul Extension, Version 1.0	55
10. "A" Standard Extension for Atomic Instructions, Version 2.1	57
10.1. Specifying Ordering of Atomic Instructions	57
10.2. Load-Reserved/Store-Conditional Instructions	57
10.3. Eventual Success of Store-Conditional Instructions	61
10.4. Atomic Memory Operations	62
11. "Zicsr", Control and Status Register (CSR) Instructions, Version 2.0	66
11.1. CSR Instructions	66
11.1.1. CSR Access Ordering	68
12. "Zicntr" and "Zihpm" Counters, Version 2.0	70
12.1. "Zicntr" Standard Extension for Base Counters and Timers	70
12.2. "Zihpm" Standard Extension for Hardware Performance Counters	72
13. "F" Standard Extension for Single-Precision Floating-Point, Version 2.2	74
13.1. F Register State	74
13.2. Floating-Point Control and Status Register	75
13.3. NaN Generation and Propagation	77
13.4. Subnormal Arithmetic	77
13.5. Single-Precision Load and Store Instructions	77
13.6. Single-Precision Floating-Point Computational Instructions	78
13.7. Single-Precision Floating-Point Conversion and Move Instructions	80
13.8. Single-Precision Floating-Point Compare Instructions	81
13.9. Single-Precision Floating-Point Classify Instruction	82
14. "D" Standard Extension for Double-Precision Floating-Point, Version 2.2	83
14.1. D Register State	83
14.2. NaN Boxing of Narrower Values	83
14.3. Double-Precision Load and Store Instructions	84
14.4. Double-Precision Floating-Point Computational Instructions	84
14.5. Double-Precision Floating-Point Conversion and Move Instructions	85
14.6. Double-Precision Floating-Point Compare Instructions	86
14.7. Double-Precision Floating-Point Classify Instruction	86
15. "Q" Standard Extension for Quad-Precision Floating-Point, Version 2.2	88
15.1. Quad-Precision Load and Store Instructions	88
15.2. Quad-Precision Computational Instructions	88
15.3. Quad-Precision Convert and Move Instructions	89
15.4. Quad-Precision Floating-Point Compare Instructions	90
15.5. Quad-Precision Floating-Point Classify Instruction	90
16. "Zfhh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point, Version 1.0	91
16.1. Half-Precision Load and Store Instructions	91
16.2. Half-Precision Computational Instructions	91
16.3. Half-Precision Conversion and Move Instructions	92
16.4. Half-Precision Floating-Point Compare Instructions	93

16.5. Half-Precision Floating-Point Classify Instruction	93
16.6. "Zfhmin" Standard Extension for Minimal Half-Precision Floating-Point	94
17. RVWMO Memory Consistency Model, Version 2.0	95
17.1. Definition of the RVWMO Memory Model	95
17.1.1. Memory Model Primitives	95
17.1.2. Syntactic Dependencies	97
17.1.3. Preserved Program Order	98
17.1.4. Memory Model Axioms	99
Load Value Axiom	99
Atomicity Axiom	99
Progress Axiom	100
17.2. CSR Dependency Tracking Granularity	100
17.3. Source and Destination Register Listings	100
18. "C" Standard Extension for Compressed Instructions, Version 2.0	107
18.1. Overview	107
18.2. Compressed Instruction Formats	109
18.3. Load and Store Instructions	110
18.3.1. Stack-Pointer-Based Loads and Stores	110
18.3.2. Register-Based Loads and Stores	113
18.4. Control Transfer Instructions	114
18.5. Integer Computational Instructions	115
18.5.1. Integer Constant-Generation Instructions	115
18.5.2. Integer Register-Immediate Operations	115
18.5.3. Integer Register-Register Operations	117
18.5.4. Defined Illegal Instruction	118
18.5.5. NOP Instruction	119
18.5.6. Breakpoint Instruction	119
18.6. Usage of C Instructions in LR/SC Sequences	119
18.7. HINT Instructions	119
18.8. RVC Instruction Set Listings	121
19. "Zimop" May-Be-Operations Extension, Version 0.1	125
19.1. "Zcmop" Compressed May-Be-Operations Extension, Version 0.2	126
20. "B" Standard Extension for Bit Manipulation, Version 0.0	127
21. "J" Standard Extension for Dynamically Translated Languages, Version 0.0	128
22. "P" Standard Extension for Packed-SIMD Instructions, Version 0.2	129
23. "V" Standard Extension for Vector Operations, Version 1.0	130
24. "Zfinx", "Zdinx", "Zhinx", "Zhinxmin": Standard Extensions for Floating-Point in Integer Registers, Version 1.0	131
24.1. Processing of Narrower Values	131
24.2. Zdinx	131
24.3. Processing of Wider Values	132
24.4. Zhinx	132
24.5. Zhinxmin	132

24.6. Privileged Architecture Implications	133
25. "Zfa" Standard Extension for Additional Floating-Point Instructions, Version 1.0	134
25.1. Load-Immediate Instructions	134
25.2. Minimum and Maximum Instructions	135
25.3. Round-to-Integer Instructions	136
25.4. Modular Convert-to-Integer Instruction	136
25.5. Move Instructions	137
25.6. Comparison Instructions	137
26. "Ztso" Standard Extension for Total Store Ordering, Version 1.0	139
27. Base Cache Management Operation ISA Extensions	140
27.1. Pseudocode for instruction semantics	140
27.2. Introduction	140
27.3. Background	141
27.3.1. Memory and Caches	141
27.3.2. Cache-Block Operations	141
27.4. Coherent Agents and Caches	142
27.4.1. Memory Ordering	143
Preserved Program Order	143
Load Values	144
27.4.2. Traps	144
Illegal Instruction and Virtual Instruction Exceptions	144
Page Fault, Guest-Page Fault, and Access Fault Exceptions	145
Address Misaligned Exceptions	146
Breakpoint Exceptions and Debug Mode Entry	146
Hypervisor Extension	147
27.4.3. Effects on Constrained LR/SC Loops	147
27.4.4. Software Discovery	148
27.5. Control and Status Register State	148
27.6. Extensions	151
27.6.1. Cache-Block Management Instructions	151
27.6.2. Cache-Block Zero Instructions	152
27.6.3. Cache-Block Prefetch Instructions	152
27.7. Instructions	153
27.7.1. cbo.clean	153
27.7.2. cbo.flush	154
27.7.3. cbo.inval	154
27.7.4. cbo.zero	155
27.7.5. prefetch.i	155
27.7.6. prefetch.r	156
27.7.7. prefetch.w	156
28. "Zawrs" Standard extension for Wait-on-Reservation-Set instructions, Version 1.01	158
28.1. Wait-on-Reservation-Set Instructions	158
29. "Zc*" Standard Extension for Code Size Reduction	160

29.1. Zc* Overview	160
29.2. C	161
29.3. Zce	161
29.4. MISA.C	162
29.5. Zca	162
29.6. Zcf (RV32 only)	162
29.7. Zcd	162
29.8. Zcb	162
29.9. Zcmp	164
29.10. Zcmt	165
29.11. Zc instruction formats	165
29.12. Zcb instructions	166
29.12.1. c.lbu	166
29.12.2. c.lhu	167
29.12.3. c.lh	168
29.12.4. c.sb	169
29.12.5. c.sh	170
29.12.6. c.zext.b	171
29.12.7. c.sext.b	172
29.12.8. c.zext.h	173
29.12.9. c.sext.h	174
29.12.10. c.zext.w	175
29.12.11. c.not	176
29.12.12. c.mul	177
29.13. PUSH/POP register instructions	178
29.13.1. PUSH/POP functional overview	178
29.13.2. Example usage	179
Stack pointer adjustment handling	180
Register list handling	180
29.13.3. PUSH/POP Fault handling	181
29.13.4. Software view of execution	181
Software view of the PUSH sequence	181
Software view of the POP/POPRET sequence	182
29.13.5. Non-idempotent memory handling	183
29.13.6. Example RV32I PUSH/POP sequences	184
cm.push {ra, s0-s2}, -64	184
cm.push {ra, s0-s11}, -112	184
cm.pop {ra}, 16	185
cm.pop {ra, s0-s3}, 48	185
cm.pop {ra, s0-s4}, 64	185
29.13.7. cm.push	186
29.13.8. cm.pop	191
29.13.9. cm.popretz	196

29.13.10. cm.popret	201
29.13.11. cm.mvsa01	206
29.13.12. cm.mva01s	207
29.14. Table Jump Overview	208
29.14.1. JVT	208
29.14.2. Table Jump Fault handling	208
29.14.3. JVT CSR	210
29.14.4. cm.jt	212
29.14.5. cm.jalt	214
30. RV32/64G Instruction Set Listings	216
31. Extending RISC-V	230
31.1. Extension Terminology	230
31.1.1. Standard versus Non-Standard Extension	230
31.1.2. Instruction Encoding Spaces and Prefixes	230
31.1.3. Greenfield versus Brownfield Extensions	231
31.1.4. Standard-Compatible Global Encodings	232
31.1.5. Guaranteed Non-Standard Encoding Space	232
31.2. RISC-V Extension Design Philosophy	232
31.3. Extensions within fixed-width 32-bit instruction format	233
31.3.1. Available 30-bit instruction encoding spaces	233
31.3.2. Available 25-bit instruction encoding spaces	233
31.3.3. Available 22-bit instruction encoding spaces	234
31.3.4. Other spaces	234
31.4. Adding aligned 64-bit instruction extensions	234
31.5. Supporting VLIW encodings	235
31.5.1. Fixed-size instruction group	235
31.5.2. Encoded-Length Groups	235
31.5.3. Fixed-Size Instruction Bundles	235
31.5.4. End-of-Group bits in Prefix	236
32. ISA Extension Naming Conventions	237
32.1. Case Sensitivity	237
32.2. Base Integer ISA	237
32.3. Instruction-Set Extension Names	237
32.4. Version Numbers	237
32.5. Underscores	238
32.6. Additional Standard Extension Names	238
32.7. Supervisor-level Instruction-Set Extensions	238
32.8. Hypervisor-level Instruction-Set Extensions	238
32.9. Machine-level Instruction-Set Extensions	239
32.10. Non-Standard Extension Names	239
32.11. Subset Naming Convention	239
33. History and Acknowledgments	241
33.1. "Why Develop a new ISA?" Rationale from Berkeley Group	241

33.2. History from Revision 1.0 of ISA manual	242
33.3. History from Revision 2.0 of ISA manual	244
33.4. Acknowledgments	245
33.5. History from Revision 2.1	245
33.6. Acknowledgments	246
33.7. History from Revision 2.2	246
33.8. Acknowledgments	246
33.9. History for Revision 2.3	246
33.10. Funding	246
Appendix A: RVWMO Explanatory Material, Version 0.1	248
A.1. Why RVWMO?	248
A.2. Litmus Tests	248
A.3. Explaining the RVWMO Rules	250
A.3.1. Preserved Program Order and Global Memory Order	250
A.3.2. Load value axiom	251
A.3.3. Atomicity axiom	253
A.3.4. Progress axiom	254
A.3.5. Overlapping-Address Orderings (Rules 1-3)	255
A.3.6. Fences (Rule 4)	257
A.3.7. Explicit Synchronization (Rules 5-8)	258
A.3.8. Syntactic Dependencies (Rules 9-11)	260
A.3.9. Pipeline Dependencies (Rules 12-13)	262
A.4. Beyond Main Memory	264
A.4.1. Coherence and Cacheability	264
A.4.2. I/O Ordering	265
A.5. Code Porting and Mapping Guidelines	266
A.6. Implementation Guidelines	272
A.6.1. Possible Future Extensions	274
A.7. Known Issues	275
A.7.1. Mixed-size RSW	275
Appendix B: Formal Memory Model Specifications, Version 0.1	277
B.1. Formal Axiomatic Specification in Alloy	277
B.2. Formal Axiomatic Specification in Herd	281
B.3. An Operational Memory Model	284
B.3.1. Intra-instruction Pseudocode Execution	287
B.3.2. Instruction Instance State	289
B.3.3. Hart State	290
B.3.4. Shared Memory State	290
B.3.5. Transitions	291
Fetch instruction	291
Initiate memory load operations	291
Satisfy memory load operation by forwarding from unpropagated stores	292
Satisfy memory load operation from memory	293

Complete load operations	294
Early sc fail	294
Paired sc	294
Initiate memory store operation footprints	294
Instantiate memory store operation values	294
Commit store instruction	295
Propagate store operation	295
Commit and propagate store operation of an sc	296
Late sc fail	296
Complete store operations	297
Satisfy, commit and propagate operations of an AMO	297
Commit fence	298
Register read	298
Register write	298
Pseudocode internal step	298
Finish instruction	299
B.3.6. Limitations	300

Index	301
-------	-----

Bibliography	303
--------------	-----

DRAFT

Preamble

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

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

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

DRAFT

Preface

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	2.0	Ratified
RV64E	2.0	Ratified
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
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
<i>Counters</i>	2.0	Ratified
<i>P</i>	<i>0.2</i>	<i>Draft</i>
V	1.0	Ratified
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zihintpause	2.0	Ratified
Zihintntl	1.0	Ratified
Zfa	1.0	Ratified
Zfh	1.0	Ratified
Zfhmin	1.0	Ratified
Zfinx	1.0	Ratified
Zdinx	1.0	Ratified
Zhinx	1.0	Ratified
Zhinxmin	1.0	Ratified
Zmmul	1.0	Ratified

Base	Version	Status
Ztso	1.0	Ratified

The changes in this version of the document include:

- The draft Zam extension has been removed, in favor of the definition of a misaligned atomicity granule PMA.

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	<i>Draft</i>
RV128I	1.7	<i>Draft</i>
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	<i>Draft</i>
L	0.0	<i>Draft</i>
B	0.0	<i>Draft</i>
J	0.0	<i>Draft</i>
T	0.0	<i>Draft</i>
P	0.2	<i>Draft</i>
V	0.7	<i>Draft</i>
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zam	0.1	<i>Draft</i>
Ztso	0.1	<i>Frozen</i>

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	<i>Draft</i>
RV128I	1.7	<i>Draft</i>
Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
M	2.0	Ratified
A	2.0	<i>Frozen</i>
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Ztso	0.1	<i>Frozen</i>
Counters	2.0	<i>Draft</i>
L	0.0	<i>Draft</i>
B	0.0	<i>Draft</i>
J	0.0	<i>Draft</i>
T	0.0	<i>Draft</i>
P	0.2	<i>Draft</i>
V	0.7	<i>Draft</i>
N	1.1	<i>Draft</i>

Base	Version	Status
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>

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
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

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

The major changes in this version of the document include:

- The previous version of this document was released under a Creative Commons Attribution 4.0

International License by the original authors, and this and future versions of this document will be released under the same license.

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

Preface to Document Version 2.1

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

- Numerous additions and improvements to the commentary sections.
- Separate version numbers for each chapter.
- Modification to long instruction encodings >64 bits to avoid moving the *rd* specifier in very long instruction formats.
- CSR instructions are now described in the base integer format where the counter registers are introduced, as opposed to only being introduced later in the floating-point section (and the companion privileged architecture manual).
- The **SCALL** and **SBREAK** instructions have been renamed to **ECALL** and **EBREAK**, respectively. Their encoding and functionality are unchanged.
- Clarification of floating-point NaN handling, and a new canonical NaN value.

- Clarification of values returned by floating-point to integer conversions that overflow.
- Clarification of **LR/SC** allowed successes and required failures, including use of compressed instructions in the sequence.
- A new **RV32E** base ISA proposal for reduced integer register counts, supports **MAC** extensions.
- A revised calling convention.
- Relaxed stack alignment for soft-float calling convention, and description of the **RV32E** calling convention.
- A revised proposal for the **C** compressed extension, version 1.9 .

Preface to Version 2.0

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

- The ISA has been divided into an integer base with several standard extensions.
- The instruction formats have been rearranged to make immediate encoding more efficient.
- The base ISA has been defined to have a little-endian memory system, with big-endian or bi-endian as non-standard variants.
- Load-Reserved/Store-Conditional (**LR/SC**) instructions have been added in the atomic instruction extension.
- **AMOs** and **LR/SC** can support the release consistency model.
- The **FENCE** instruction provides finer-grain memory and I/O orderings.
- An **AMO** for fetch-and-XOR (**AMOXOR**) has been added, and the encoding for **AMOSWAP** has been changed to make room.
- The **AUIPC** instruction, which adds a 20-bit upper immediate to the **PC**, replaces the **RDNPC** instruction, which only read the current **PC** value. This results in significant savings for position-independent code.
- The **JAL** instruction has now moved to the **U-Type** format with an explicit destination register, and the **J** instruction has been dropped being replaced by **JAL** with $rd=x0$. This removes the only instruction with an implicit destination register and removes the **J-Type** instruction format from the base ISA. There is an accompanying reduction in **JAL** reach, but a significant reduction in base ISA complexity.
- The static hints on the **JALR** instruction have been dropped. The hints are redundant with the rd and $rs1$ register specifiers for code compliant with the standard calling convention.
- The **JALR** instruction now clears the lowest bit of the calculated target address, to simplify hardware and to allow auxiliary information to be stored in function pointers.
- The **MFTX.S** and **MFTX.D** instructions have been renamed to **FMV.X.S** and **FMV.X.D**, respectively. Similarly, **MXTF.S** and **MXTF.D** instructions have been renamed to **FMV.S.X** and **FMV.D.X**, respectively.
- The **MFFSR** and **MTFSR** instructions have been renamed to **FRCSR** and **FSCSR**, respectively. **FRRM**, **FSRM**, **FRFLAGS**, and **FSFLAGS** instructions have been added to individually access the rounding mode and exception flags subfields of the **fcsr**.
- The **FMV.X.S** and **FMV.X.D** instructions now source their operands from $rs1$, instead of $rs2$. This

change simplifies datapath design.

- **FCLASS.S** and **FCLASS.D** floating-point classify instructions have been added.
- A simpler NaN generation and propagation scheme has been adopted.
- For **RV32I**, the system performance counters have been extended to 64-bits wide, with separate read access to the upper and lower 32 bits.
- Canonical **NOP** and **MV** encodings have been defined.
- Standard instruction-length encodings have been defined for 48-bit, 64-bit, and >64-bit instructions.
- Description of a 128-bit address space variant, **RV128**, has been added.
- Major opcodes in the 32-bit base instruction format have been allocated for user-defined custom extensions.
- A typographical error that suggested that stores source their data from *rd* has been corrected to refer to *rs2*.

DRAFT

Chapter 1. Introduction

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

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



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



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

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



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

1.1. RISC-V Hardware Platform Terminology

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

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

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

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

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

1.2. RISC-V Software Execution Environments and Harts

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

- "Bare metal" hardware platforms where harts are directly implemented by physical processor threads and instructions have full access to the physical address space. The hardware platform

defines an execution environment that begins at power-on reset.

- RISC-V operating systems that provide multiple user-level execution environments by multiplexing user-level harts onto available physical processor threads and by controlling access to memory via virtual memory.
- RISC-V hypervisors that provide multiple supervisor-level execution environments for guest operating systems.
- RISC-V emulators, such as Spike, QEMU or rv8, which emulate RISC-V harts on an underlying x86 system, and which can provide either a user-level or a supervisor-level execution environment.



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

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

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

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



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

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

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

1.3. RISC-V ISA Overview

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

Although it is convenient to speak of *the* RISC-V ISA, RISC-V is actually a family of related ISAs, of which there are currently four base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, described in [Chapter 2](#) and [Chapter 7](#), 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 6](#) 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. [Chapter 8](#) sketches a future RV128I variant of the base integer instruction set supporting a flat 128-bit address space (XLEN=128). 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, so we ensured this could be accommodated within the 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 ADD 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. [Chapter 31](#) describes various ways of extending the RISC-V ISA. We have also developed a naming convention for RISC-V base instructions and instruction-set extensions, described in detail in [Chapter 32](#).

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) be vacant, or (2) contain *main memory*, or (3) 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. 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 non-vacant 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. Vacant locations in the address space are never accessible.

Except when specified otherwise, implicit reads that do not raise an exception and that have no side effects may occur arbitrarily early and speculatively, even before the machine could possibly prove that the read will be needed. For instance, a valid implementation could attempt to read all of main memory at the earliest opportunity, cache as many fetchable (executable) bytes as possible for later instruction fetches, and avoid reading main memory for instruction fetches ever again. To ensure that certain implicit reads are ordered only after writes to the same memory locations, software must execute specific fence or cache-control instructions defined for this purpose (such as the FENCE.I instruction defined in [Chapter 3](#)).

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 17](#) and in appendices. Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in [Chapter 26](#). 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 18](#) 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.

[Table 1](#) illustrates the standard RISC-V instruction-length encoding convention. 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**.

1.5.1. Expanded Instruction-Length Encoding

A portion of the 32-bit instruction-encoding space has been tentatively allocated for instructions longer than 32 bits. The entirety of this space is reserved at this time, and the following proposal for encoding instructions longer than 32 bits is not considered frozen.

Standard instruction-set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in [Table 1](#). Instruction lengths between 80 bits and 176 bits are encoded using a 3-bit field in bits [14:12] giving the number of 16-bit words in addition to the first 5×16 -bit words. The encoding with bits [14:12] set to "111" is reserved for future longer instruction encodings.

Table 1. RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

			xxxxxxxxxxxxxxxxaa	16-bit (aa≠11)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxxxbbb11	32-bit (bbb≠111)
	...xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
	...xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	64-bit
	...xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx111111	(80+16*nnn)-bit, nnn≠111
	...xxxx	xxxxxxxxxxxxxxxx	x111xxxxx111111	Reserved for ≥192-bits
Byte Address:	base+4	base+2	base	

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. As described [Chapter 31](#), 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. Further, if the implementation also does not need instructions >32-bits in length, it can recover a further four major opcodes for non-conforming 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 2 shows the characteristics of each kind of trap.

	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

Table 2 Characteristics of traps: 1) Termination may be requested. 2) Imprecise fatal traps might be observable by software.

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

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

1.7. UNSPECIFIED Behaviors and Values

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

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

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

Chapter 2. RV32I Base Integer Instruction Set, Version 2.1

This chapter describes the RV32I base integer instruction set.



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

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

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



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



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

2.1. Programmers' Model for Base Integer ISA

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

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

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

XLEN-1	0
x0/zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	

XLEN-1	0
x9	
x10	
x11	
x12	
x13	
x14	
x15	
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x29	
x30	
x31	
XLEN	
XLEN-1	0
pc	
XLEN	

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 regfile 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 6).

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**).



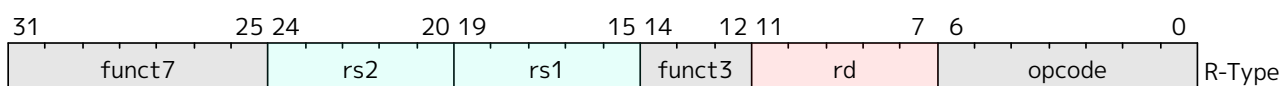
Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with **IALIGN=32**, where these are the only places where misalignment can occur.

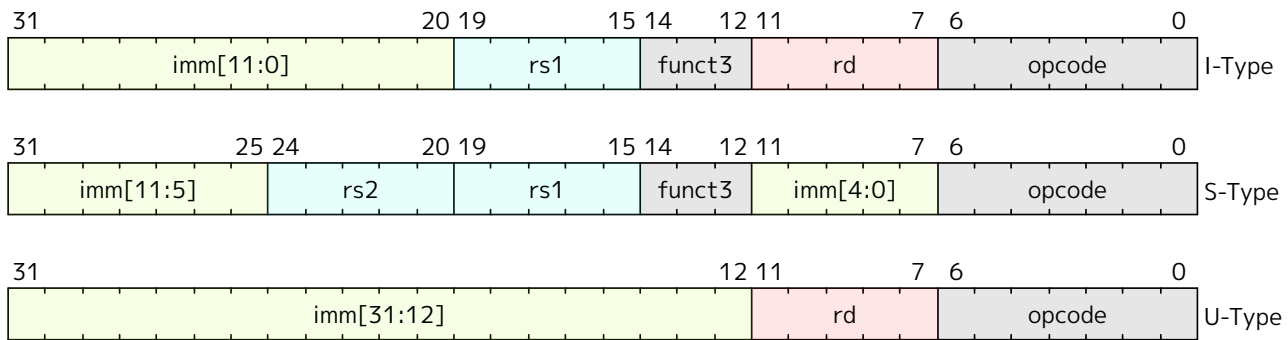
The behavior upon decoding a reserved instruction is UNSPECIFIED.



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 11), 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)).

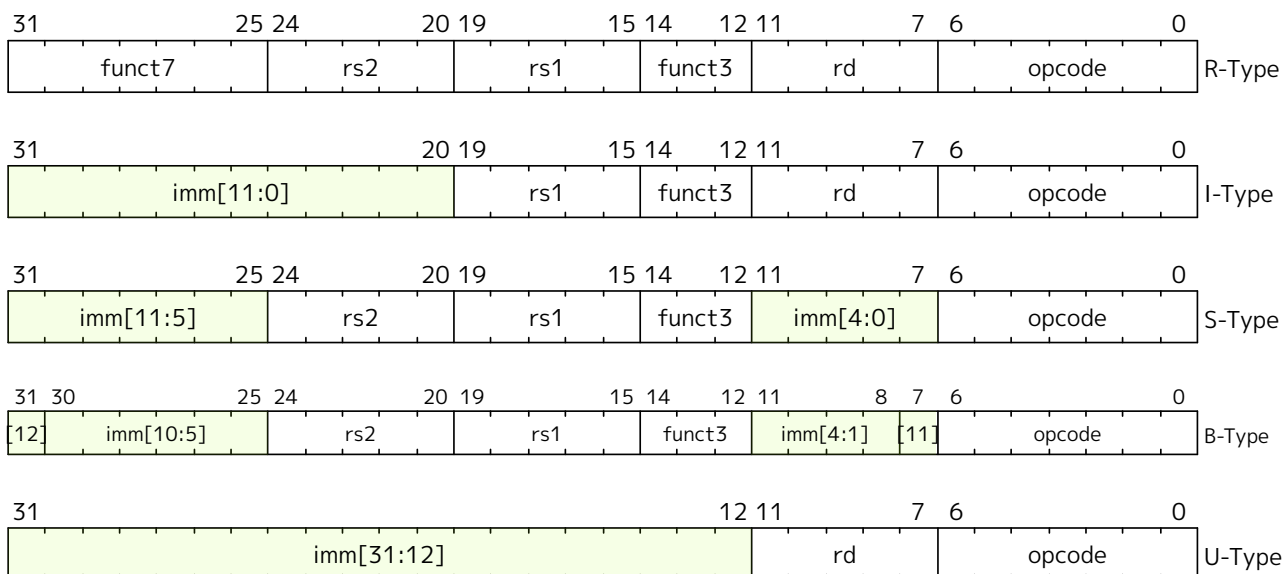


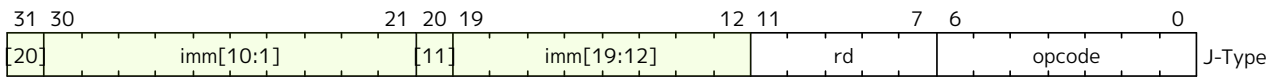
In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load-upper-immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

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.

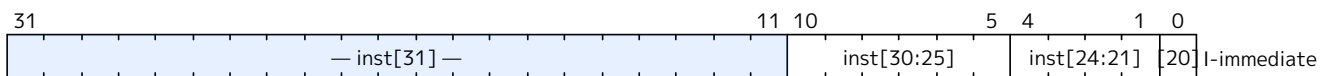
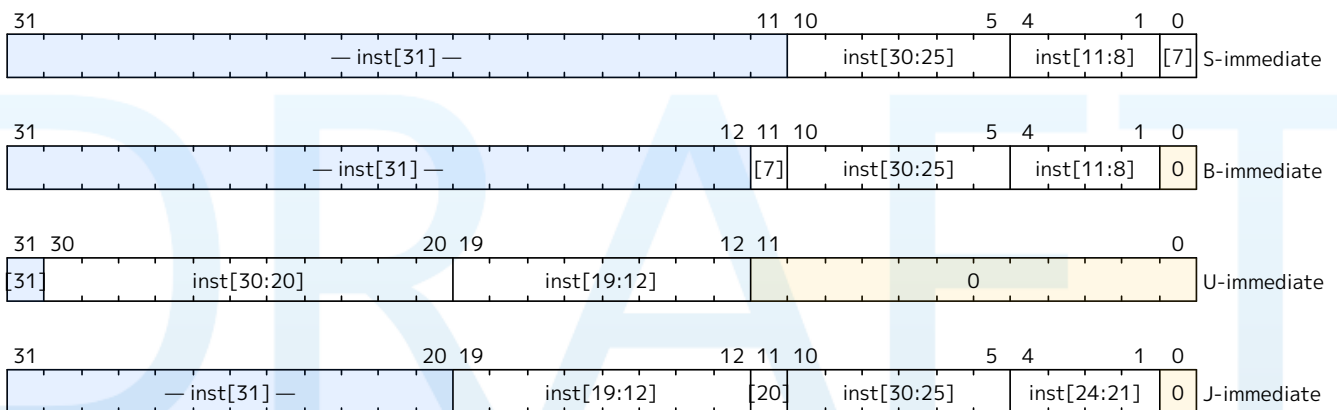


Figure 1. Types of immediate produced by RISC-V instructions.



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 muxes to multiply the immediate by 2, we reduce instruction signal fanout and immediate mux costs by around a factor of 2. The scrambled immediate encoding will add negligible time to static or ahead-of-time compilation. For dynamic generation of instructions, there is some small additional overhead, but the most common short forward branches have straightforward immediate encodings.

2.4. Integer Computational Instructions

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

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

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

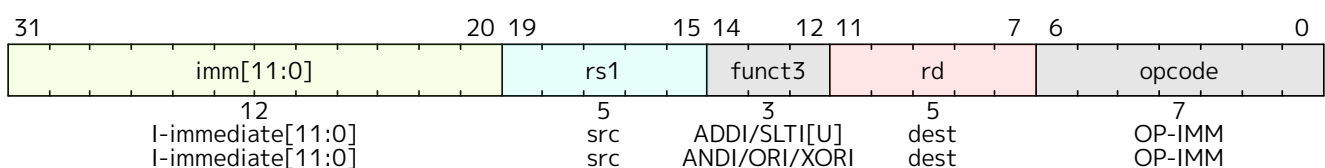


For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

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

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

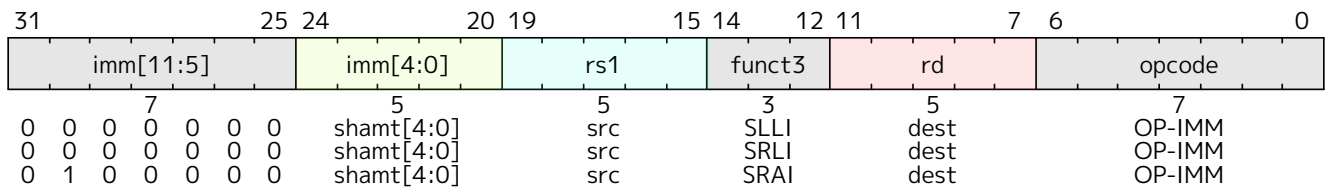
2.4.1. Integer Register-Immediate Instructions



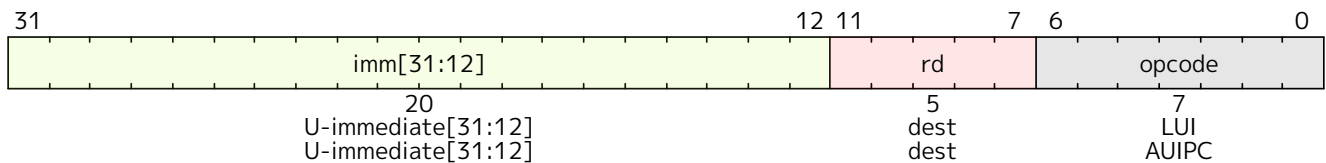
ADDI adds the sign-extended 12-bit immediate to register *rs1*. Arithmetic overflow is ignored and the result is simply the low XLEN bits of the result. ADDI *rd, rs1, 0* is used to implement the `MV rd, rs1` assembler pseudoinstruction.

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

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



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



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

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

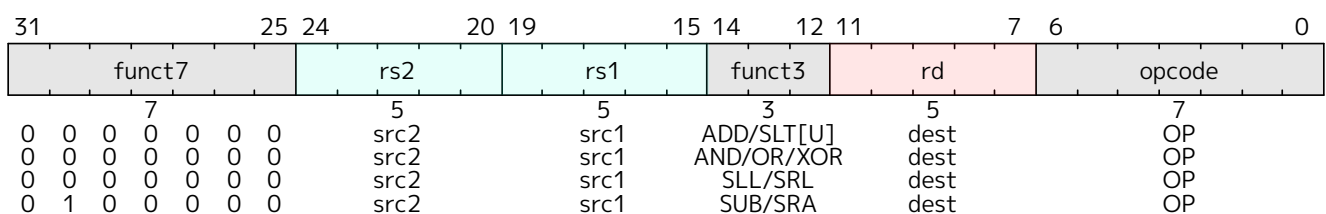
The assembly syntax for **lui** and **auipc** does not represent the lower 12 bits of the U-immediate, which are always zero.

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

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

2.4.2. Integer Register-Register Operations

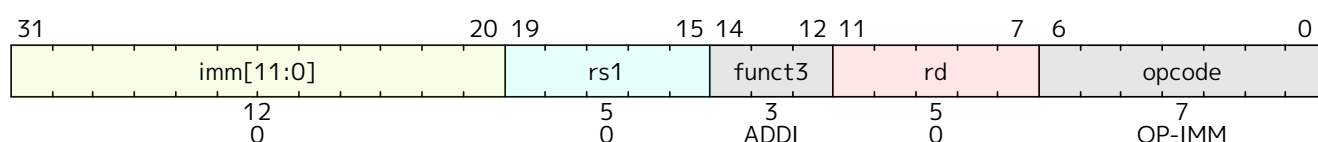
RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



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

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

2.4.3. NOP Instruction



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



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

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

2.5. Control Transfer Instructions

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

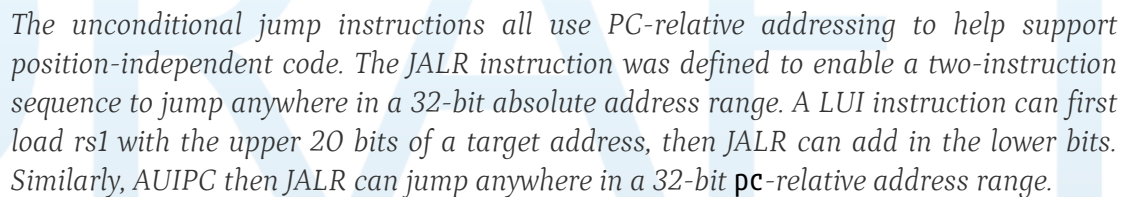
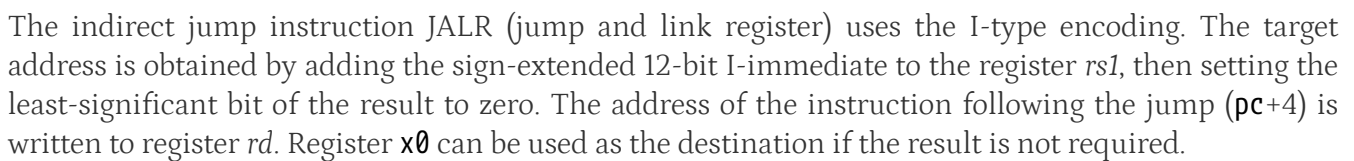
If an instruction access-fault or instruction page-fault exception occurs on the target of a jump or taken branch, the exception is reported on the target instruction, not on the jump or branch instruction.

2.5.1. Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed



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



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 RISC-V Instruction Set Manual Volume I | © RISC-V

extensions with 16-bit aligned instructions, such as 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.

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

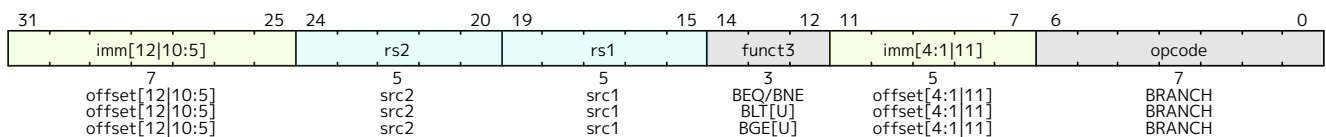
Some other ISAs added explicit hint bits to their indirect-jump instructions to guide return-address stack manipulation. We use implicit hinting tied to register numbers and the calling convention to reduce the encoding space used for these hints.



*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 that support extensions with 16-bit aligned instructions, such as 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.

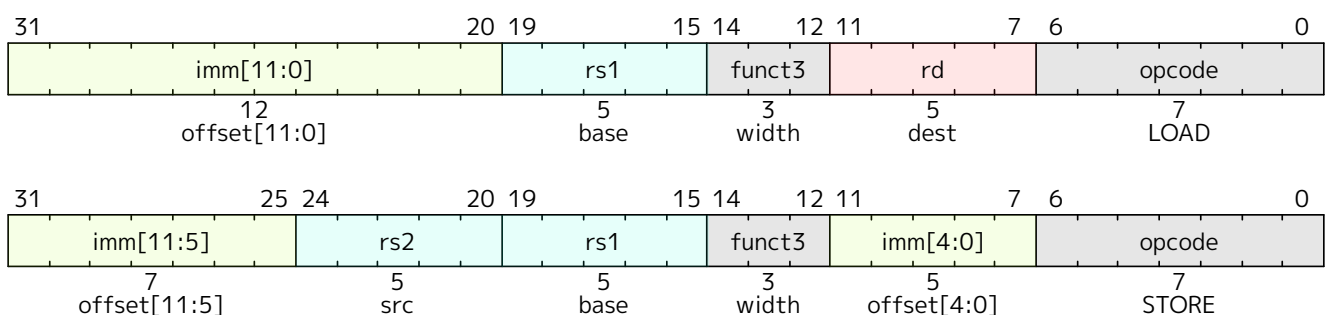
The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.

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).

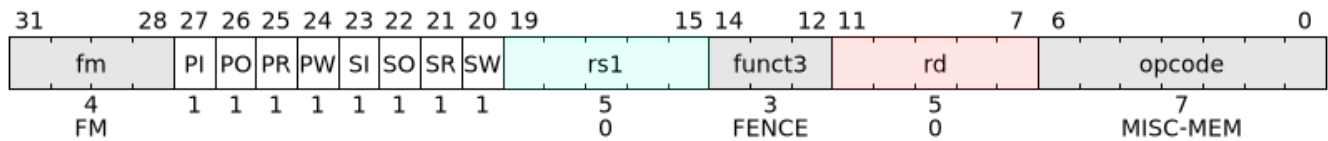
Even when misaligned loads and stores complete successfully, these accesses might run extremely slowly depending on the implementation (e.g., when implemented via an invisible trap). Furthermore, whereas naturally aligned loads and stores are guaranteed to execute atomically, misaligned loads and stores 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



The FENCE instruction is 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 17](#) provides a precise description of the RISC-V memory consistency model.

The FENCE instruction also orders memory reads and writes made by the hart as observed by memory reads and writes made by an external device. However, FENCE does 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 the FENCE instruction 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.

Table 4. Fence mode encoding

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

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

The 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 in [Table 4](#) are also reserved for future use. Base implementations shall treat all such reserved configurations as normal fences with fm=0000, and standard software shall use only non-reserved configurations.



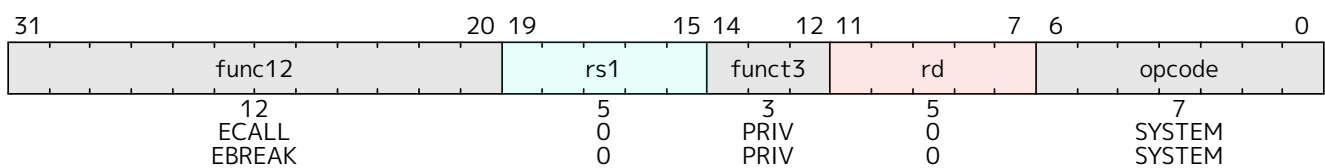
We chose a relaxed memory model to allow high performance from simple machine implementations and from likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative 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 11](#), and the base unprivileged instructions are described in the following section.



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



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

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

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



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



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

Another use of EBREAK is to support "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       # NOP encoding the semihosting call number 7
```

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 18](#).

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}	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}	
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
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}	
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

Instruction	Constraints	Code Points	Purpose
SLTI	$rd \neq x0$	2^{17}	Designated for custom use
SLTIU	$rd \neq x0$	2^{17}	
SLLI	$rd \neq x0$	2^{10}	
SRLI	$rd \neq x0$	2^{10}	
SRAI	$rd \neq x0$	2^{10}	
SLT	$rd \neq x0$	2^{10}	
SLTU	$rd \neq x0$	2^{10}	

DRAFT

Chapter 3. "Zifencei" 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.

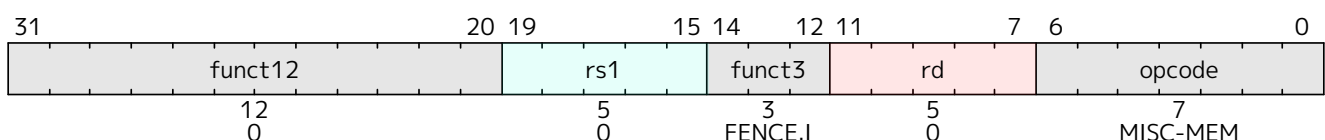
The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) 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.

The unused fields in the FENCE.I instruction, *imm*[11:0], *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.

DRAFT

Chapter 4. "Zihintntl" 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 6 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 6. 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

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



The working-set sizes listed in [Table 6](#) 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 7](#) 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 extension is provided, compressed variants of these HINTs are also provided: C.NTL.P1 is encoded as C.ADD $x0, x2$; C.NTL.PALL is encoded as C.ADD $x0, x3$; C.NTL.S1 is encoded as C.ADD $x0, x4$; and C.NTL.ALL is encoded as C.ADD $x0, x5$.

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



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

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

Table 7. 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 reexecuted 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 5. "Zihintpause" 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.

A future extension might add primitives similar to the x86 MONITOR/MWAIT instructions, which provide a more efficient mechanism to wait on writes to a specific memory location. However, these instructions 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.

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 6. RV32E and RV64E Base Integer Instruction Sets, Version 2.0

This chapter describes a proposal for 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 7](#).



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.

6.1. 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.

6.2. 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 7. 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.

7.1. Register State

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

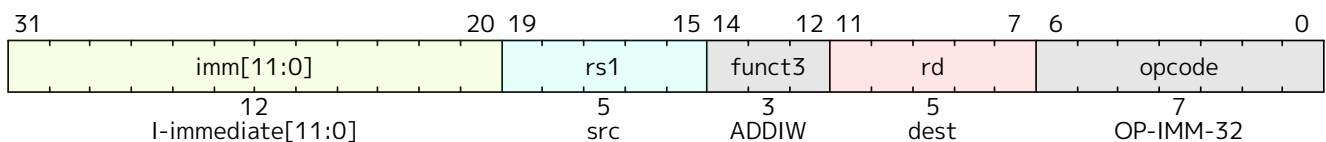
7.2. Integer Computational Instructions

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.

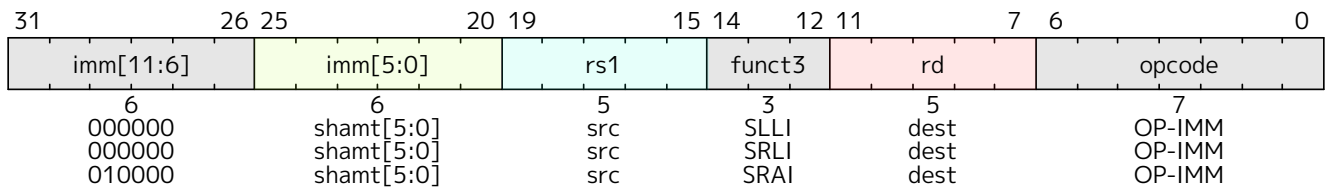


The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion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

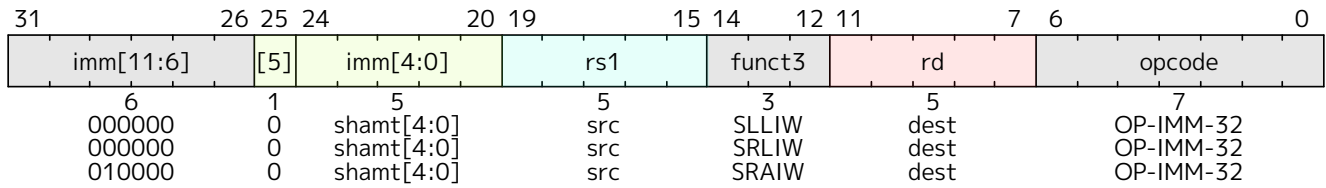
7.2.1. Integer Register-Immediate Instructions



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



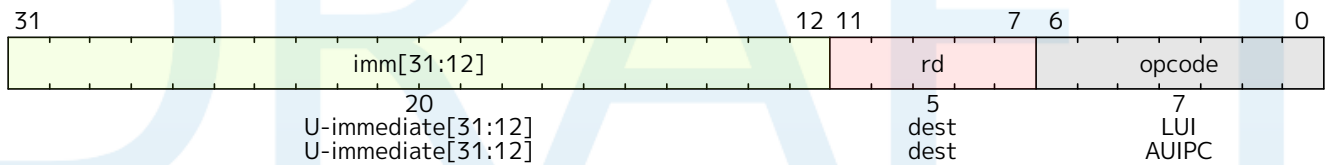
Shifts by a constant are encoded as a specialization of the I-type format using the same instruction opcode as RV32I. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the I-immediate field for RV64I. The right shift type is encoded in bit 30. SLLI is a logical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



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



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



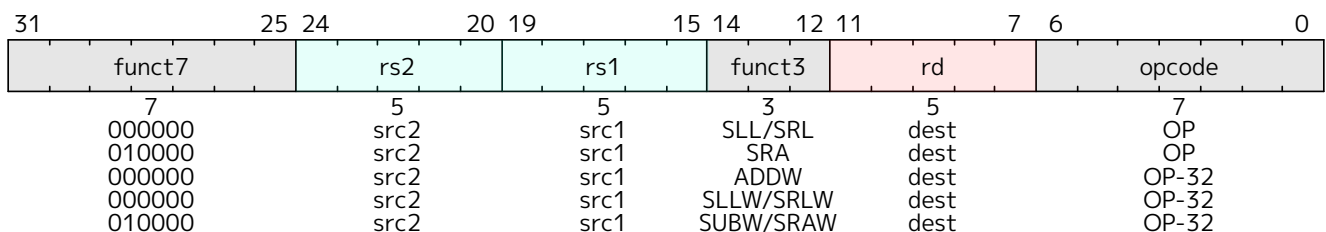
LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 32-bit U-immediate into register *rd*, filling in the lowest 12 bits with zeros. The 32-bit result is sign-extended to 64 bits.

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



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

7.2.2. Integer Register-Register Operations



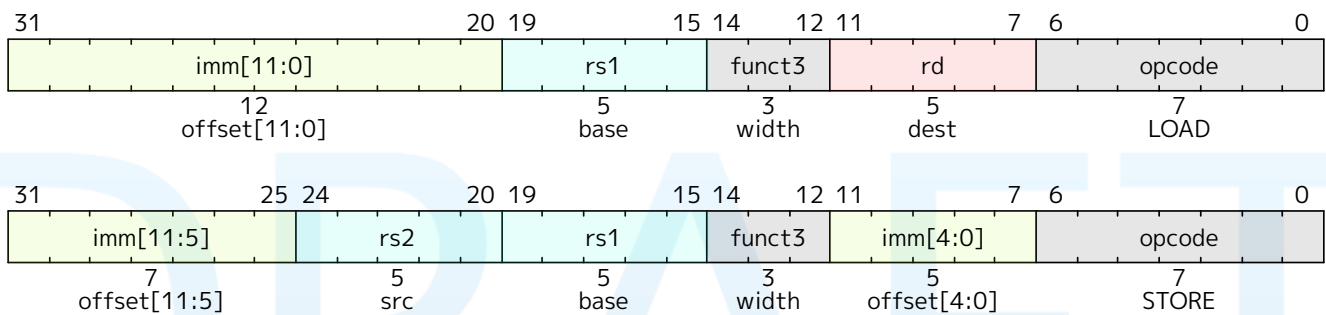
ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

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].

7.3. 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.

7.4. 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 8](#) lists all RV64I HINT code points. 91% of the HINT space is reserved for standard HINTs, but none are presently defined. The remainder of the HINT space is designated for custom HINTs; no standard HINTs will ever be defined in this subspace.

Table 8. RV64I 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}	
ADDIW	$rd=x0$	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
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}	
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	
FENCE	$pred=0$ or $succ=0$, $pred \neq W, succ=0$	15	
FENCE	$rd=rs1=x0, fm=0$, $pred=W, succ=0$	1	PAUSE

Instruction	Constraints	Code Points	Purpose
SLTI	$rd=x0$	2^{17}	<i>Designated for custom use</i>
SLTIU	$rd=x0$	2^{17}	
SLLI	$rd=x0$	2^{11}	
SRLI	$rd=x0$	2^{11}	
SRAI	$rd=x0$	2^{11}	
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}	

DRAFT

Chapter 8. RV128I Base Integer Instruction Set, Version 1.7

There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management.

— Bell and Strecker, ISCA-3, 1976.

This chapter describes RV128I, a variant of the RISC-V ISA supporting a flat 128-bit address space. The variant is a straightforward extrapolation of the existing RV32I and RV64I designs.



The primary reason to extend integer register width is to support larger address spaces. It is not clear when a flat address space larger than 64 bits will be required. At the time of writing, the fastest supercomputer in the world as measured by the Top500 benchmark had over 1PB of DRAM, and would require over 50 bits of address space if all the DRAM resided in a single address space. Some warehouse-scale computers already contain even larger quantities of DRAM, and new dense solid-state non-volatile memories and fast interconnect technologies might drive a demand for even larger memory spaces. Exascale systems research is targeting 100PB memory systems, which occupy 57 bits of address space. At historic rates of growth, it is possible that greater than 64 bits of address space might be required before 2030. History suggests that whenever it becomes clear that more than 64 bits of address space is needed, architects will repeat intensive debates about alternatives to extending the address space, including segmentation, 96-bit address spaces, and software workarounds, until, finally, flat 128-bit address spaces will be adopted as the simplest and best solution. We have not frozen the RV128 spec at this time, as there might be need to evolve the design based on actual usage of 128-bit address spaces.

RV128I builds upon RV64I in the same way RV64I builds upon RV32I, with integer registers extended to 128 bits (i.e., XLEN=128). Most integer computational instructions are unchanged as they are defined to operate on XLEN bits. The RV64I `"*W"` integer instructions that operate on 32-bit values in the low bits of a register are retained but now sign extend their results from bit 31 to bit 127. A new set of `"*D"` integer instructions are added that operate on 64-bit values held in the low bits of the 128-bit integer registers and sign extend their results from bit 63 to bit 127. The `"*D"` instructions consume two major opcodes (OP-IMM-64 and OP-64) in the standard 32-bit encoding.



To improve compatibility with RV64, in a reverse of how RV32 to RV64 was handled, we might change the decoding around to rename RV64I ADD as a 64-bit ADDD, and add a 128-bit ADDQ in what was previously the OP-64 major opcode (now renamed the OP-128 major opcode).

Shifts by an immediate (SLLI/SRLI/SRAI) are now encoded using the low 7 bits of the I-immediate, and variable shifts (SLL/SRL/SRA) use the low 7 bits of the shift amount source register.

A LDU (load double unsigned) instruction is added using the existing LOAD major opcode, along with new LQ and SQ instructions to load and store quadword values. SQ is added to the STORE major opcode, while LQ is added to the MISC-MEM major opcode.

The floating-point instruction set is unchanged, although the 128-bit Q floating-point extension can now support FMV.X.Q and FMV.Q.X instructions, together with additional FCVT instructions to and

from the T (128-bit) integer format.

DRAFT

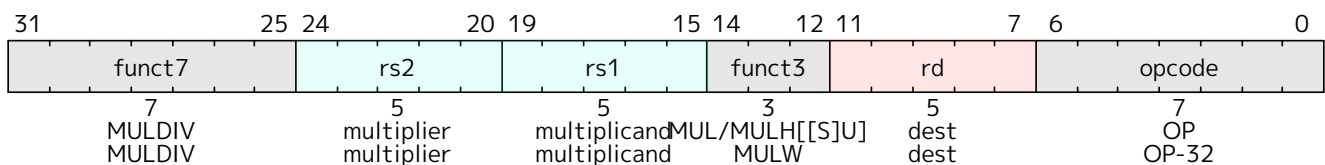
Chapter 9. "M" Standard Extension for Integer Multiplication and Division, Version 2.0

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.

9.1. 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, respectively. 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.



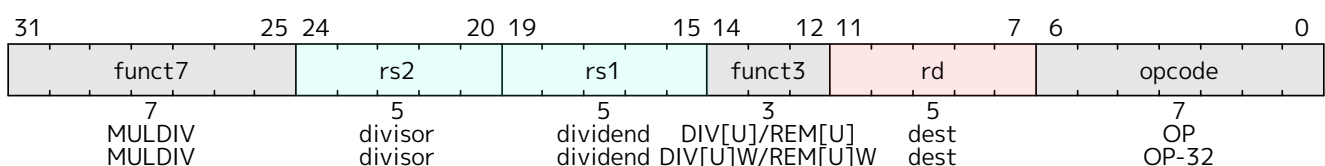
MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the multiplicand (which contains the sign bit) with the less-significant words of the multiplier (which are unsigned).

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].

9.2. 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 respectively, 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 respectively. 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 9. 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 9. 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 implementors 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.

9.3. 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 9.1](#), 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.*

DRAFT

Chapter 10. "A" Standard Extension for Atomic Instructions, Version 2.1

The standard 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.

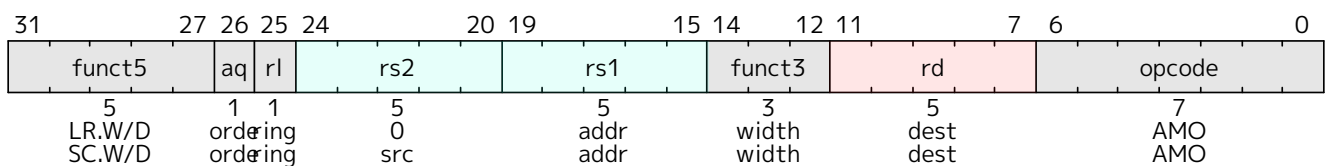
10.1. Specifying Ordering of Atomic Instructions

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.

10.2. 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*. 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 mux 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 A 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 17.1](#).

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 17.1](#) 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.

DRAFT

```

# 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.

```

Example 1. 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 [Example 1](#). If inlined, compare-and-swap functionality need only take four instructions.

10.3. Eventual Success of Store-Conditional Instructions

The standard A 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. If the "C" extension is supported, then compressed forms of the aforementioned "I" instructions 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.

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.



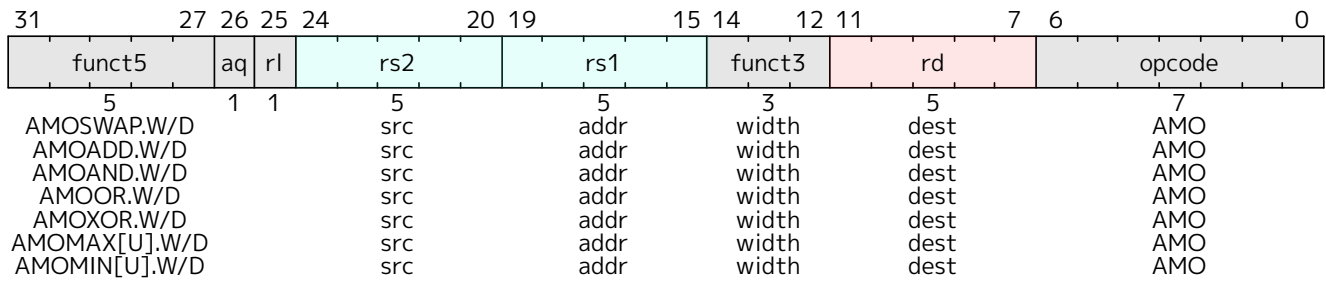
Loads and load-reserved instructions do not by themselves impede the progress of other harts' LR/SC sequences. We note this constraint implies, among other things, that loads and load-reserved instructions executed by other harts (possibly within the same core) cannot impede LR/SC progress indefinitely. 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.

10.4. Atomic Memory Operations



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

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

The misaligned atomicity granule PMA, defined in Volume II of this manual, optionally relaxes this alignment requirement. If present, the misaligned atomicity granule PMA specifies the size of a misaligned atomicity granule, a power-of-two number of bytes. The misaligned atomicity granule PMA applies only to AMOs, loads and stores defined in the base ISAs, and loads and stores of no more than XLEN bits defined in the F, D, and Q extensions. 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.

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 [Example 2](#). 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.

DRAFT

```

    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.

```

*Example 2. Sample code for mutual exclusion. **a0** contains the address of the lock.*



We recommend the use of the AMO Swap idiom shown above 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 11. "Zicsr", 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.

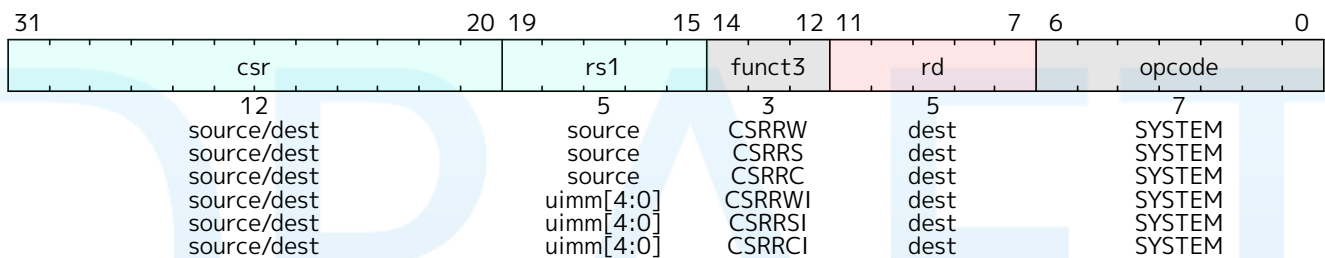


While CSRs are primarily used by the privileged architecture, there are several uses in unprivileged code including for counters and timers, and for floating-point status.

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.

11.1. 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.

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 holding a zero value other than **x0**, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects. A CSRRW with *rs1*=**x0** will attempt to write zero to the destination CSR.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned

immediate (`uimm[4:0]`) field encoded in the `rs1` field instead of a value from an integer register. For CSRRSI and CSRRCI, if the `uimm[4:0]` field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal-instruction exceptions on accesses to read-only CSRs. For CSRRWI, if `rd=x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read. Both CSRRSI and CSRRCI will always read the CSR and cause any read side effects regardless of `rd` and `rs1` fields.

Table 10. Conditions determining whether a CSR instruction reads or writes the specified CSR.

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

Table 10 summarizes the behavior of the CSR instructions with respect to whether they read and/or write the CSR.

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



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

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 write is done instead of the increment. In particular, a value written to **instret** by one instruction will be the value read by the following instruction.

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

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

11.1.1. 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 17](#)), CSR accesses are weakly ordered by default, so other harts or devices may observe CSR accesses in an order different from program order. In addition, CSR accesses are not ordered with respect to explicit memory accesses, unless a CSR access modifies the execution behavior of the instruction that performs the explicit memory access or unless a CSR access and an explicit memory access are ordered by either the syntactic dependencies defined by the memory model or the ordering requirements defined by the Memory-Ordering PMAs section in Volume II of this manual. To enforce ordering in all other cases, software should execute a **FENCE** instruction between the relevant accesses. For the purposes of the **FENCE** instruction, CSR read accesses are classified as device input (I), and CSR write accesses are classified as device output (O).



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

*These CSR-ordering constraints are imposed to support ordering main memory and memory-mapped I/O accesses with respect to CSR accesses that are visible to, or affected by, devices or other harts. Examples include the **time**, **cycle**, and **mcycle** CSRs, in addition to CSRs that reflect pending interrupts, like **mip** and **sip**. Note that implicit reads of such CSRs (e.g., taking an interrupt because of a change in **mip**) are also ordered as device input.*

*Most CSRs (including, e.g., the **fcsr**) are not visible to other harts; their accesses can be freely reordered in the global memory order with respect to FENCE instructions without violating this specification.*

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



The rules for the reordering of CSR accesses in the global memory order should probably be moved to [Chapter 17](#) concerning the RVWMO memory consistency model.

DRAFT

Chapter 12. "Zicntr" and "Zihpm" 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.

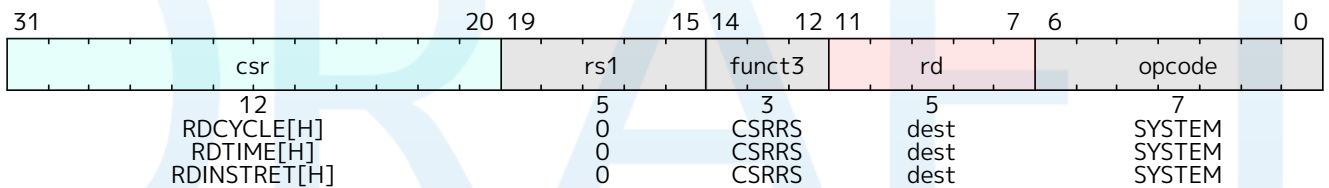
12.1. "Zicntr" Standard Extension for Base Counters and Timers

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. For a low-end implementation, the upper 32 bits of each counter can be implemented using software counters incremented by a trap handler triggered by overflow of the lower 32 bits. 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 implementor 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.

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
```

Example 3. Sample code for reading the 64-bit cycle counter when XLEN=32.

12.2. "Zihpm" Standard 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, is platform-specific. Accessing an unimplemented or ill-configured counter may cause an illegal-instruction exception or 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 13. "F" Standard 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 ([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.

13.1. F Register State

The F extension adds 32 floating-point registers, **f0-f31**, each 32 bits wide, and a floating-point control and status register **fcsr**, which contains the operating mode and exception status of the floating-point unit. This additional state is shown in [Table 11](#). 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 regfile 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 11. 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

FLEN-1	0
f16	
f17	
f18	
f19	
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	
FLEN	
31	0
fcsr	
32	

13.2. 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](#).

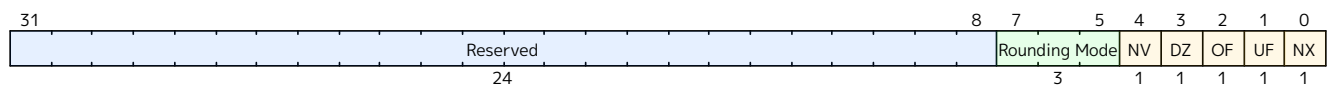


Figure 2. Floating-point control and status register

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 12. 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 12. 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 13. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 13. 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.

13.3. 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.

Implementors 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.

13.4. Subnormal Arithmetic

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

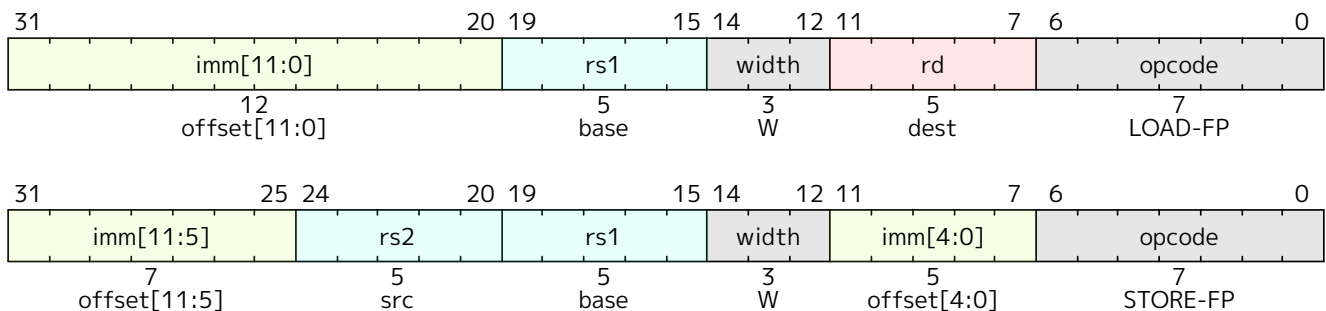
In the parlance of the IEEE standard, tininess is detected after rounding.



Detecting tininess after rounding results in fewer spurious underflow signals.

13.5. 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.

13.6. 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 14](#). It is set to S (00) for all instructions in the F extension.

Table 14. 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

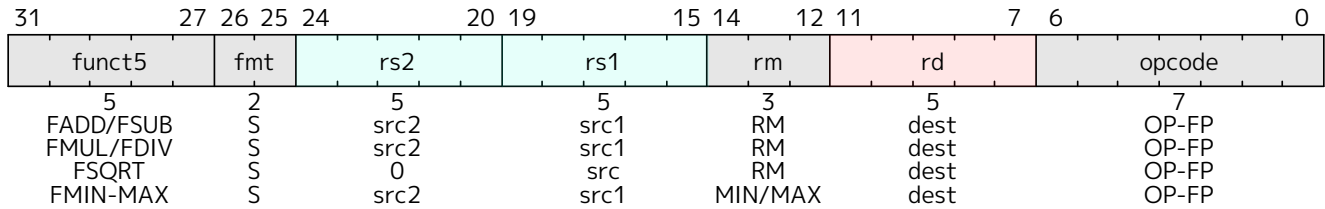
All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in [Table 12](#).

Floating-point minimum-number and maximum-number instructions FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. For the purposes of these instructions only, the value -0.0 is considered to be less than the value $+0.0$. If both inputs are NaNs, the result is the

canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.



Note that in version 2.2 of the F extension, the FMIN.S and FMAX.S instructions were amended to implement the proposed IEEE 754-201x minimumNumber and maximumNumber operations, rather than the IEEE 754-2008 minNum and maxNum operations. These operations differ in their handling of signaling NaNs.



Floating-point fused multiply-add instructions require a new standard instruction format. R4-type instructions specify three source registers (*rs1*, *rs2*, and *rs3*) and a destination register (*rd*). This format is only used by the floating-point fused multiply-add instructions.

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

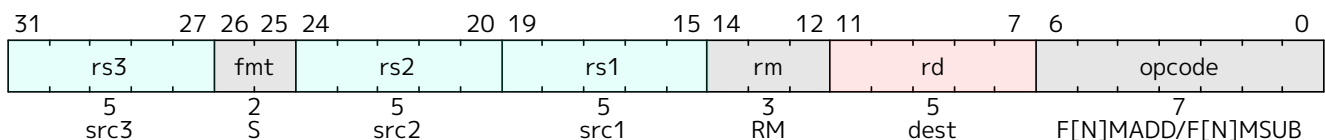
FMSUB.S multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. FMSUB.S computes $(rs1 \times rs2) - rs3$.

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

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



The FNMSUB and FNMADD instructions are counterintuitively named, owing to the naming of the corresponding instructions in MIPS-IV. The MIPS instructions were defined to negate the sum, rather than negating the product as the RISC-V instructions do, so the naming scheme was more rational at the time. The two definitions differ with respect to signed-zero results. The RISC-V definition matches the behavior of the x86 and ARM fused multiply-add instructions, but unfortunately the RISC-V FNMSUB and FNMADD instruction names are swapped compared to x86 and 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 $\infty \times 0 + qNaN$.

13.7. Single-Precision Floating-Point Conversion and Move Instructions

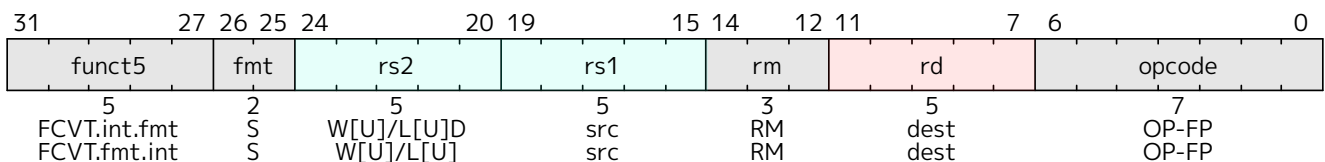
Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.S or FCVT.L.S converts a floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.S.W or FCVT.S.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a floating-point number in floating-point register *rd*. FCVT.WU.S, FCVT.LU.S, FCVT.S.WU, and FCVT.S.LU variants convert to or from unsigned integer values. For $XLEN > 32$, FCVT.W[U].S sign-extends the 32-bit result to the destination register width. FCVT.L[U].S and FCVT.S.L[U] are RV64-only instructions. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. Table 15 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.

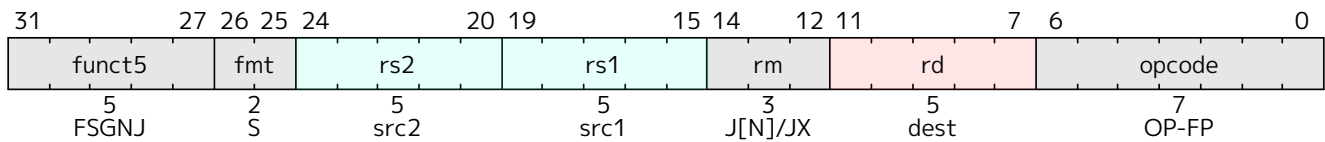
Table 15. Domains of float-to-integer conversions and behavior for invalid inputs

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



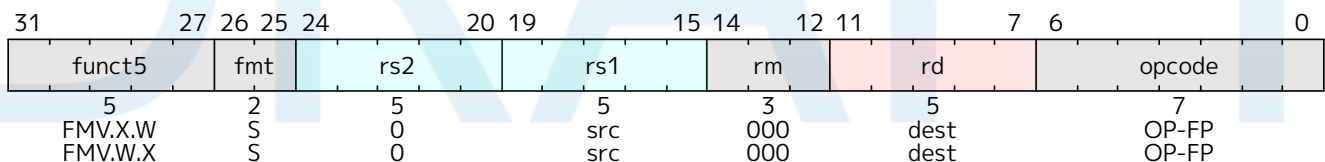
The sign-injection instructions provide floating-point MV, ABS, and NEG, as well as supporting a few other operations, including the IEEE copySign operation and sign manipulation in transcendental math function libraries. Although MV, ABS, and NEG only need a single register operand, whereas FSGNJ instructions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



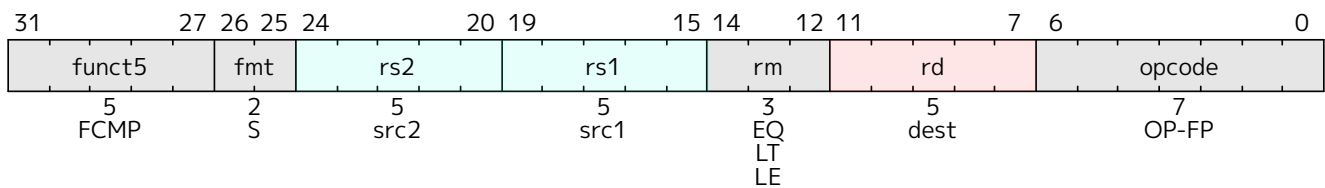
The base floating-point ISA was defined so as to allow implementations to employ an internal recoding of the floating-point format in registers to simplify handling of subnormal values and possibly to reduce functional unit latency. To this end, the F extension avoids representing integer values in the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

13.8. Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers ($=$, $<$, \leq) 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.

13.9. 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 16. 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.

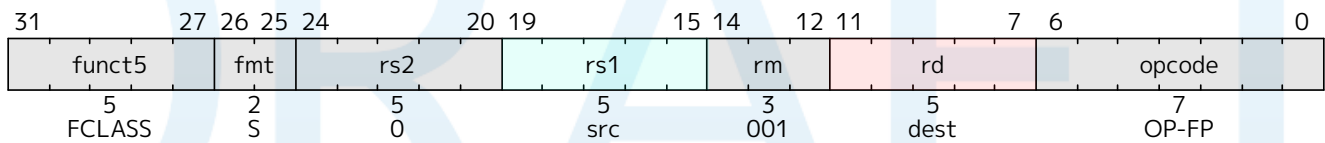


Table 16. Format of result of FCLASS instruction.

rd bit	Meaning
0	<i>rs1</i> is $-\infty$.
1	<i>rs1</i> is a negative normal number.
2	<i>rs1</i> is a negative subnormal number.
3	<i>rs1</i> is -0 .
4	<i>rs1</i> is $+0$.
5	<i>rs1</i> is a positive subnormal number.
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.

Chapter 14. "D" Standard 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.

14.1. D Register State

The D extension widens the 32 floating-point registers, **f0-f31**, to 64 bits (FLEN=64 in [Table 11](#). The **f** registers can now hold either 32-bit or 64-bit floating-point values as described below in [Section 14.2](#).



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.

14.2. 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 varargs, 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 document did not define the behavior of feeding the results of narrower or wider operands into an operation, except to require that wider saves and restores would preserve the value of a narrower operand. The new definition removes this

implementation-specific behavior, while still accommodating both non-recoded and recoded implementations of the floating-point unit. The new definition also helps catch software errors by propagating NaNs if values are used incorrectly.

Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in writing all 1s to the upper bits of a result.

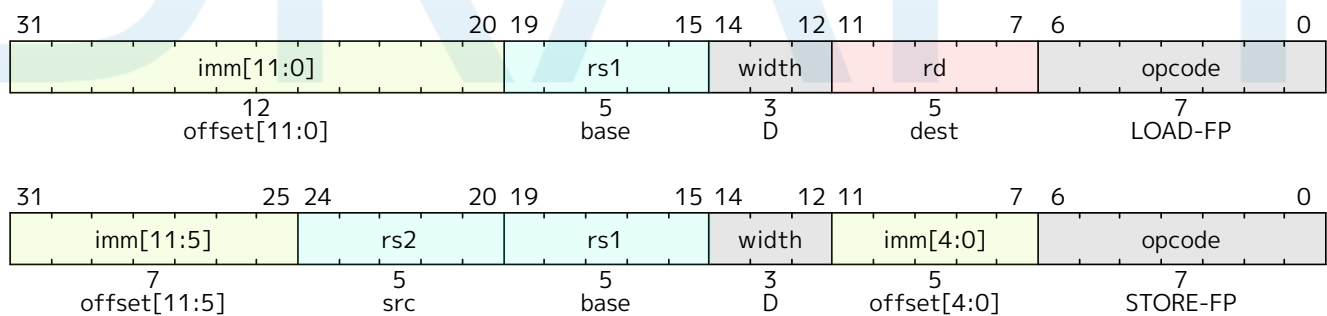
Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can be done without adding new state bits by recoding NaNs internally in the exponent field. Small modifications are needed to the pipelines used to transfer values in and out of the recoded format, but the datapath and latency costs are minimal. The recoding process has to handle shifting of input subnormal values for wide operands in any case, and extracting the NaN-boxed value is a similar process to normalization except for skipping over leading-1 bits instead of skipping over leading-0 bits, allowing the datapath muxing to be shared.

14.3. 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.

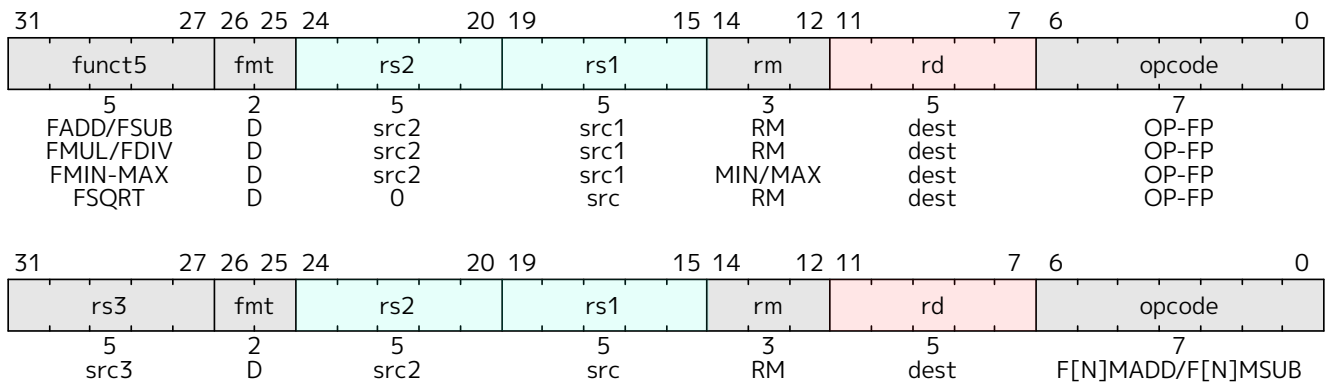


FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and $XLEN \geq 64$.

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

14.4. Double-Precision Floating-Point Computational Instructions

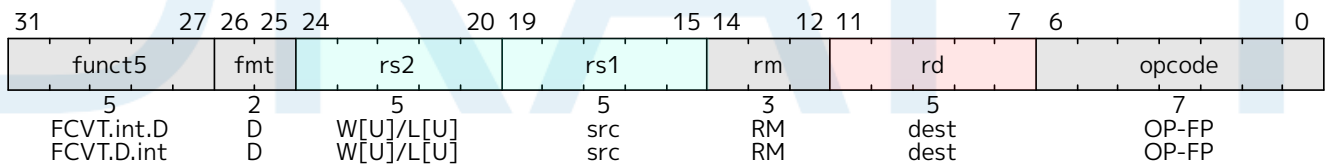
The double-precision floating-point computational instructions are defined analogously to their single-precision counterparts, but operate on double-precision operands and produce double-precision results.



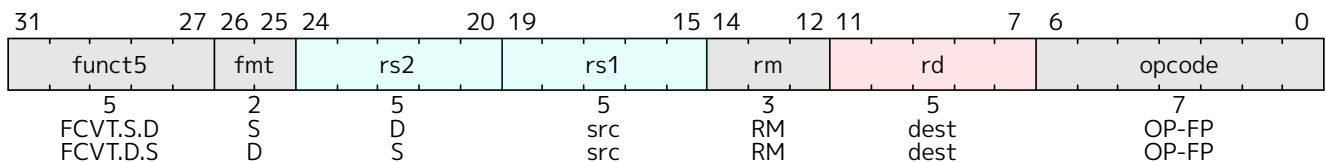
14.5. Double-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.D or FCVT.L.D converts a double-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.D.W or FCVT.D.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a double-precision floating-point number in floating-point register *rd*. FCVT.WU.D, FCVT.LU.D, FCVT.D.WU, and FCVT.D.LU variants convert to or from unsigned integer values. For RV64, FCVT.W[U].D sign-extends the 32-bit result. FCVT.L[U].D and FCVT.D.L[U] are RV64-only instructions. The range of valid inputs for FCVT.int.D and the behavior for invalid inputs are the same as for FCVT.int.S.

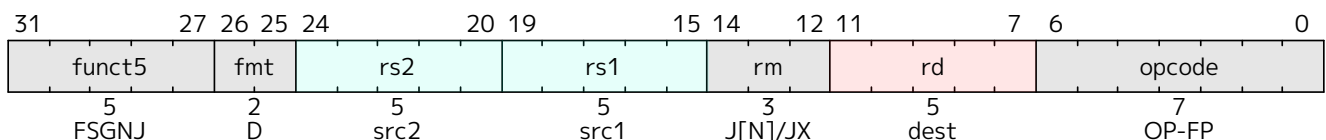
All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. Note FCVT.D.W[U] always produces an exact result and is unaffected by rounding mode.



The double-precision to single-precision and single-precision to double-precision conversion instructions, FCVT.S.D and FCVT.D.S, are encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The *rs2* field encodes the datatype of the source, and the *fmt* field encodes the datatype of the destination. FCVT.S.D rounds according to the RM field; FCVT.D.S will never round.

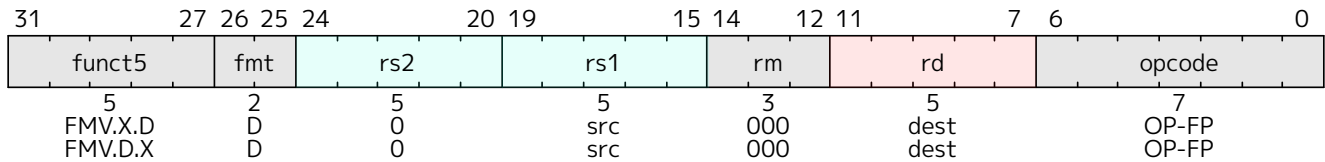


Floating-point to floating-point sign-injection instructions, FSGNJ.D, FSGNJN.D, and FSGNJX.D are defined analogously to the single-precision sign-injection instruction.



For $XLEN \geq 64$ 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.

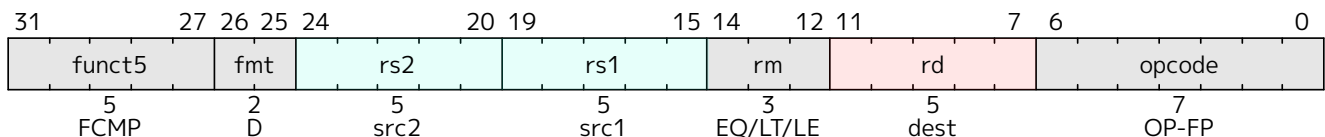


Early versions of the RISC-V ISA had additional instructions to allow RV32 systems to transfer between the upper and lower portions of a 64-bit floating-point register and an integer register. However, these would be the only instructions with partial register writes and would add complexity in implementations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

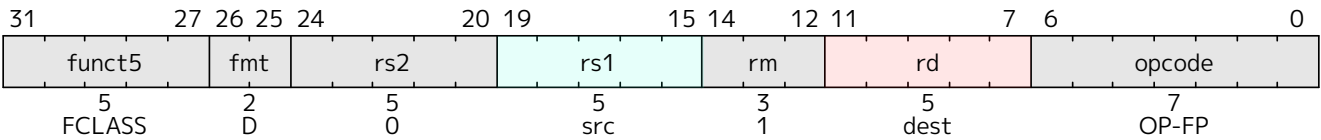
14.6. Double-Precision Floating-Point Compare Instructions

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



14.7. 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.



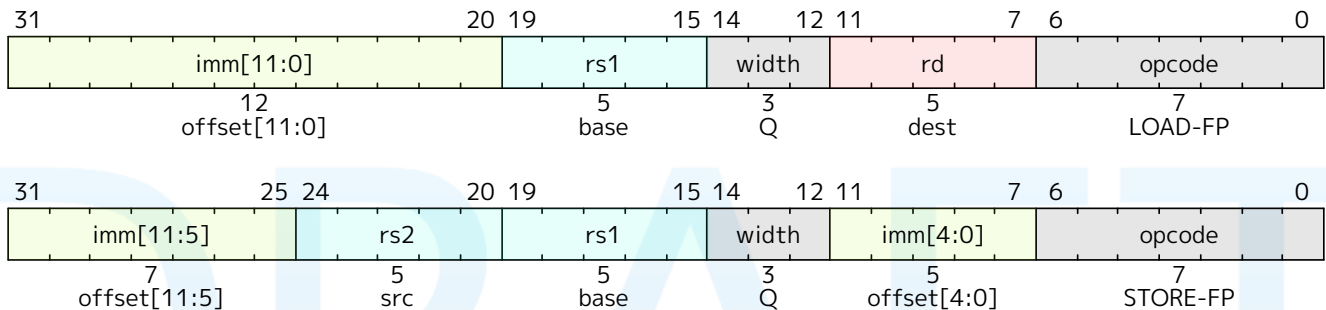
DRAFT

Chapter 15. "Q" Standard 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 14.2](#) 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.

15.1. 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.

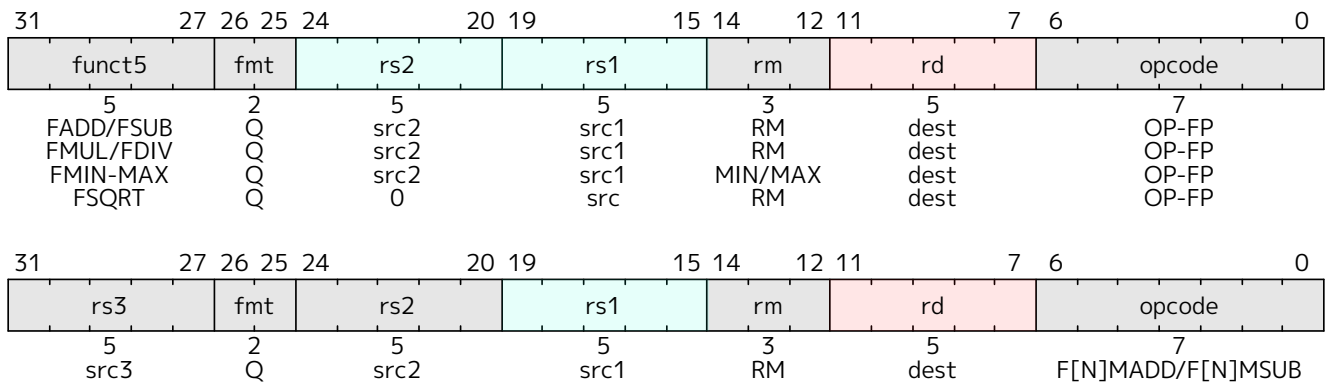
15.2. Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in [Table 17](#)

Table 17. Format field encoding.

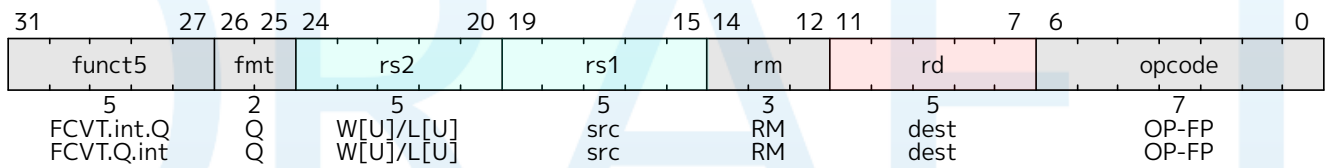
<i>fmt</i> field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

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.

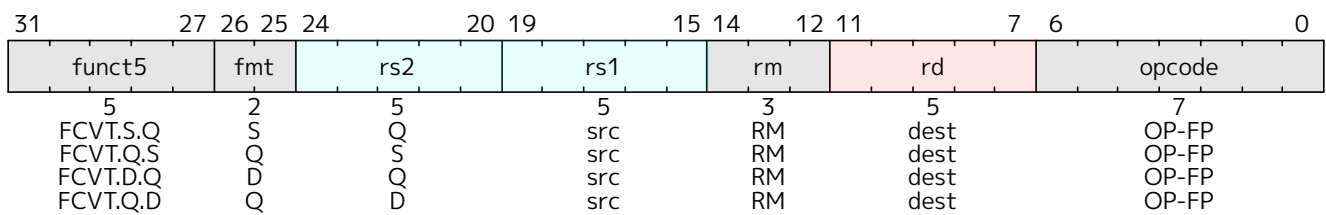


15.3. Quad-Precision Convert and Move Instructions

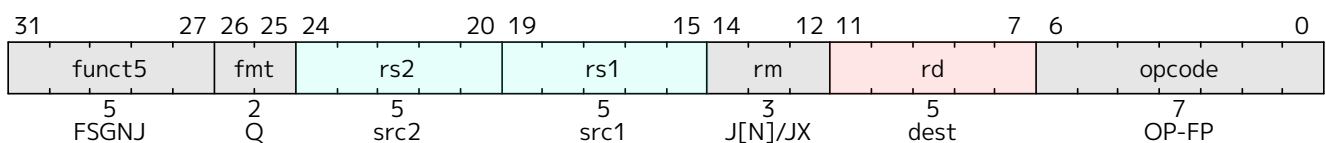
New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision-to-integer and integer-to-double-precision conversion instructions. FCVT.W.Q or FCVT.L.Q converts a quad-precision floating-point number to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

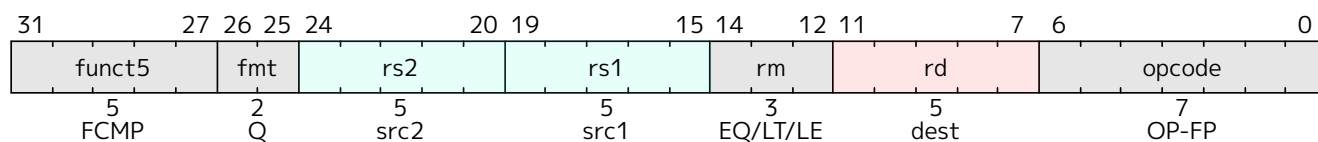


FMV.X.Q and FMV.Q.X instructions are not provided in RV32 or RV64, so quad-precision bit patterns must be moved to the integer registers via memory.

*RV128 will support FMV.X.Q and FMV.Q.X in the Q extension.*

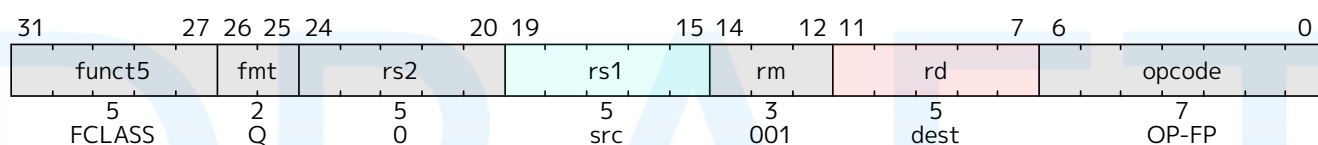
15.4. Quad-Precision Floating-Point Compare Instructions

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



15.5. Quad-Precision Floating-Point Classify Instruction

The quad-precision floating-point classify instruction, FCLASS.Q, is defined analogously to its double-precision counterpart, but operates on quad-precision operands.



Chapter 16. "Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point, Version 1.0

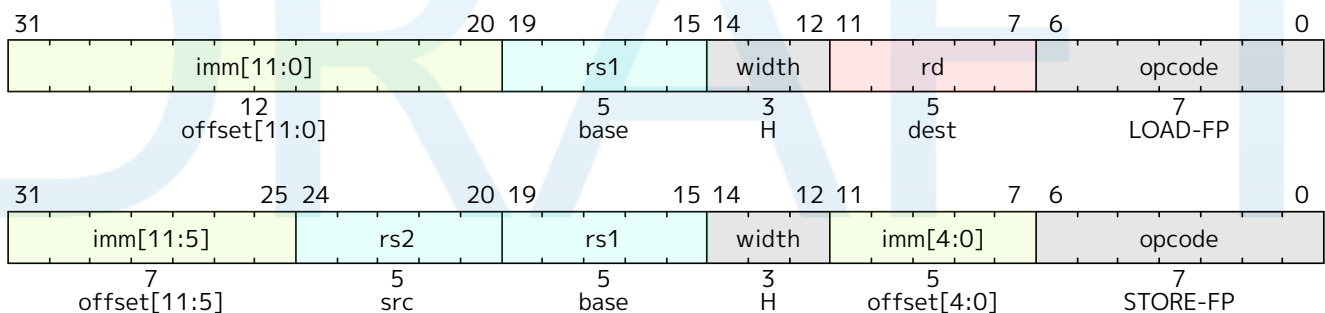
This chapter describes the Zfh standard extension for 16-bit half-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The Zfh extension depends on the single-precision floating-point extension, F. The NaN-boxing scheme described in [Section 14.2](#) is extended 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., $\text{half} + \text{single} \rightarrow \text{half}$.

16.1. 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`.

16.2. Half-Precision Computational Instructions

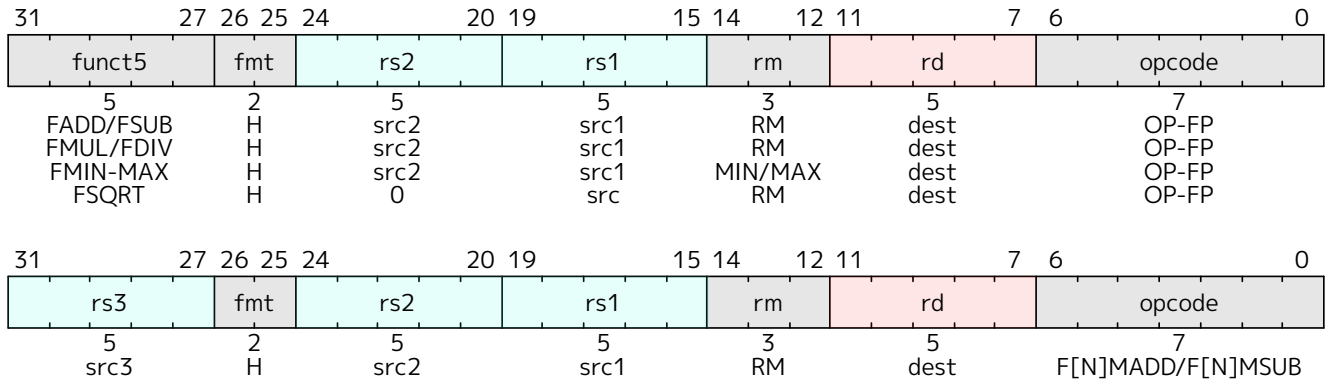
A new supported format is added to the format field of most instructions, as shown in [Table 18](#).

Table 18. Format field encoding.

<i>fmt</i> field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision

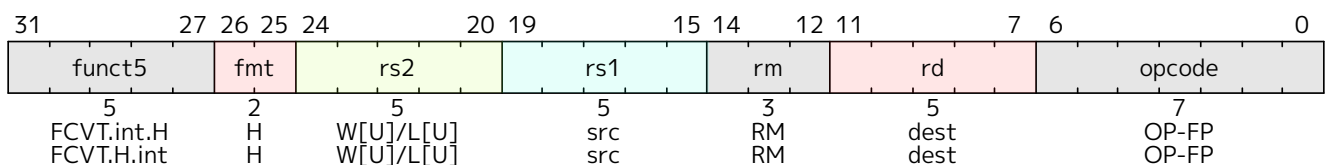
<i>fmt</i> field	Mnemonic	Meaning
10	H	16-bit half-precision
11	Q	128-bit quad-precision

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.

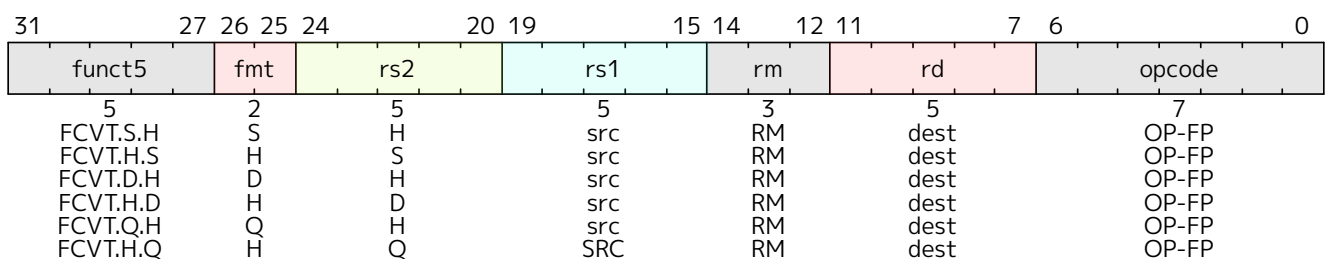


16.3. Half-Precision Conversion and Move Instructions

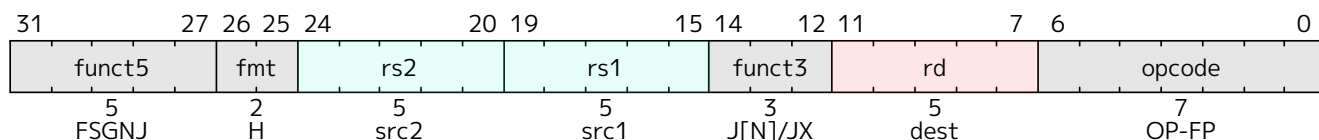
New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the single-precision-to-integer and integer-to-single-precision conversion instructions. FCVT.W.H or FCVT.L.H converts a half-precision floating-point number to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



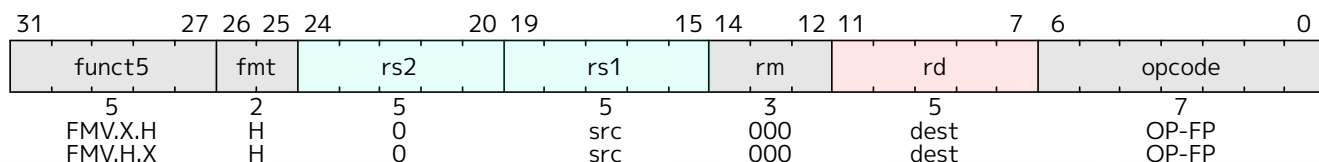
Floating-point to floating-point sign-injection instructions, FSGNJ.H, FSGNJN.H, and FSGNJX.H are defined analogously to the single-precision sign-injection instruction.



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.

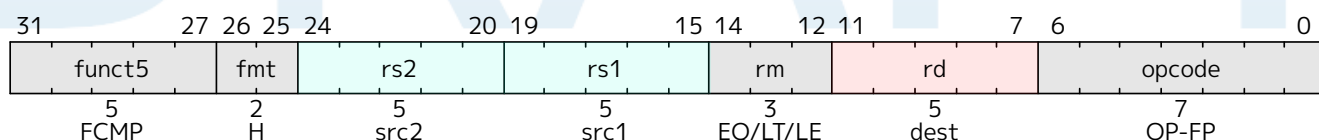
FMV.H.X moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register *rs1* to the floating-point register *rd*, NaN-boxing the result.

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



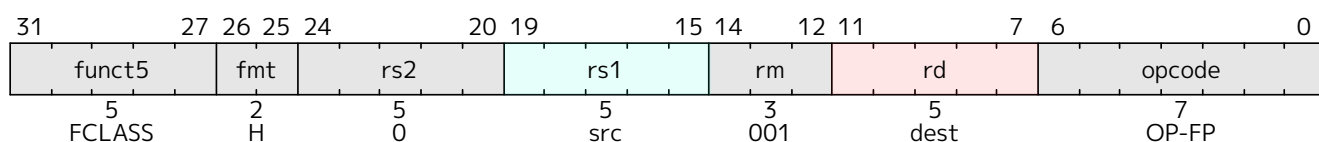
16.4. Half-Precision Floating-Point Compare Instructions

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



16.5. 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.



16.6. "Zfhmin" Standard Extension for Minimal Half-Precision Floating-Point

This section describes the Zfhmin standard extension, which provides minimal support for 16-bit half-precision binary floating-point instructions. The Zfhmin extension is a subset of the Zfh extension, consisting only of data transfer and conversion instructions. Like Zfh, the Zfhmin extension depends on the single-precision floating-point extension, F. The expectation is that Zfhmin software primarily uses the half-precision format for storage, performing most computation in higher precision.

The Zfhmin extension includes the following instructions from the Zfh extension: FLH, FSH, FMV.X.H, FMV.H.X, FCVT.S.H, and FCVT.H.S. If the D extension is present, the FCVT.D.H and FCVT.H.D instructions are also included. If the Q extension is present, the FCVT.Q.H and FCVT.H.Q instructions are additionally included.

Zfhmin does not include the FSGNJ.H instruction, because it suffices to instead use the FSGNJ.S instruction to move half-precision values between floating-point registers.

Half-precision addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the half-precision operands to single-precision, performing the operation using single-precision arithmetic, then converting back to half-precision. (Roux, 2014) Performing half-precision fused multiply-addition using this method incurs a 1-ulp error on some inputs for the RNE and RMM rounding modes.

Conversion from 8- or 16-bit integers to half-precision can be emulated by first converting to single-precision, then converting to half-precision. Conversion from 32-bit integer can be emulated by first converting to double-precision. If the D extension is not present and a 1-ulp error under RNE or RMM is tolerable, 32-bit integers can be first converted to single-precision instead. The same remark applies to conversions from 64-bit integers without the Q extension.



Chapter 17. 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 26](#)) 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. The RV128 base ISA and 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.

17.1. 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).

17.1.1. 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.

Among instructions in RV32GC and RV64GC, each aligned memory instruction gives rise to exactly one memory operation, with two exceptions. First, an unsuccessful SC instruction does not give rise to any memory operations. Second, FLD and FSD instructions may each give rise to multiple memory operations if $XLEN < 64$, as stated in [Section 14.3](#) and clarified below. An aligned AMO gives rise to a single memory operation that is both a load operation and a store operation simultaneously.



Instructions in the RV128 base instruction set and in future ISA extensions such as V (vector) and 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. An FLD or FSD instruction for which $XLEN < 64$ 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 10.2](#).

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 26](#)). 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.

17.1.2. 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 17.2](#).

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 17.3](#) 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 17.3](#)
- 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 17.3](#)
- 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 17.3](#).

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.

17.1.3. 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

17.1.4. 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*.

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

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.

Progress Axiom

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

17.2. CSR Dependency Tracking Granularity

Table 19. 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.

17.3. 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 17.1.2](#).

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 20. 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>		
ADD	<i>rs1, rs2</i>	<i>rd</i>		
SUB	<i>rs1, rs2</i>	<i>rd</i>		
SLL	<i>rs1, rs2</i>	<i>rd</i>		
SLT	<i>rs1, rs2</i>	<i>rd</i>		
SLTU	<i>rs1, rs2</i>	<i>rd</i>		
XOR	<i>rs1, rs2</i>	<i>rd</i>		
SRL	<i>rs1, rs2</i>	<i>rd</i>		
SRA	<i>rs1, rs2</i>	<i>rd</i>		

	Source Registers	Destination Registers	Accumulating CSRs	
OR	$rs1, rs2$	rd		
AND	$rs1, rs2$	rd		
FENCE				
FENCE.I				
ECALL				
EBREAK				
CSRRW [‡]	$rs1, csr^*$	rd, csr		* unless $rd=x0$
CSRRS [‡]	$rs1, csr$	rd^*, csr		* unless $rs1=x0$
CSRRC [‡]	$rs1, csr$	rd^*, csr		* unless $rs1=x0$
[‡] carries a dependency from $rs1$ to csr and from csr to rd				
CSRRWI [‡]	csr^*	rd, csr		* unless $rd=x0$
CSRRSI [‡]	csr	rd, csr^*		* unless $uimm[4:0]=0$
CSRRCI [‡]	csr	rd, csr^*		* unless $uimm[4:0]=0$

Table 21. 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		
SRLW	$rs1, rs2$	rd		
SRAW	$rs1, rs2$	rd		

Table 22. RV32M Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
MUL	$rs1, rs2$	rd		
MULH	$rs1, rs2$	rd		

	Source Registers	Destination Registers	Accumulating CSRs	
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 23. 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 24. 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 25. 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>		
AMOADD.D [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOXOR.D [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		
AMOAND.D [†]	<i>rs1</i> ^A , <i>rs2</i> ^D	<i>rd</i>		

	Source Registers	Destination Registers	Accumulating CSRs	
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 26. 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 27. RV64F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.S	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NV, NX	[*] if <i>rm</i> =111
FCVT.LU.S	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NV, NX	[*] if <i>rm</i> =111
FCVT.S.L	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NX	[*] if <i>rm</i> =111
FCVT.S.LU	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NX	[*] if <i>rm</i> =111

Table 28. RV32D Standard Extension

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

Table 29. RV64D Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
FCVT.L.D	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NV, NX	[*] if <i>rm</i> =111
FCVT.LU.D	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NV, NX	[*] if <i>rm</i> =111
FMV.X.D	<i>rs1</i>	<i>rd</i>		
FCVT.D.L	<i>rs1</i> , <i>frm</i> [*]	<i>rd</i>	NX	[*] if <i>rm</i> =111
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 18. "C" Standard 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 (RV32, RV64, RV128), 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.

18.1. 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, RV64C, and RV128C, but as shown in [Table 34](#), a few opcodes are used for different purposes depending on base ISA. For example, the wider address-space RV64C and RV128C variants require 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. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. 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 and RV128C.



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, RV64I/E, or RV128I) 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)

18.2. Compressed Instruction Formats

Table 30 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 31 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 muxes 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 30. 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 31. Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, CA, and CB formats.

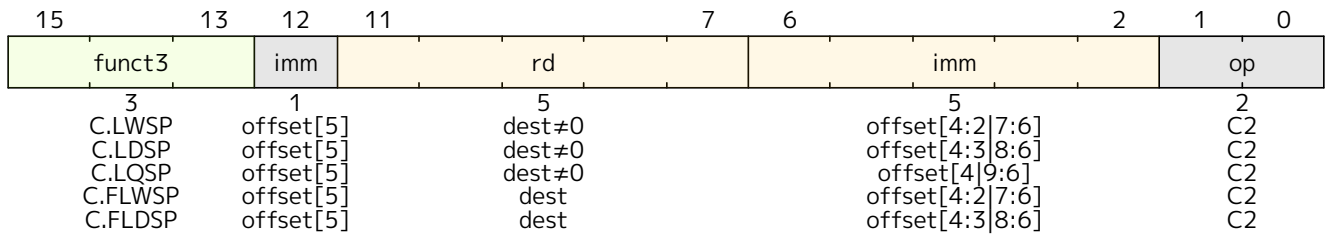
RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

18.3. 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.

18.3.1. Stack-Pointer-Based Loads and Stores



These instructions use the CI format.

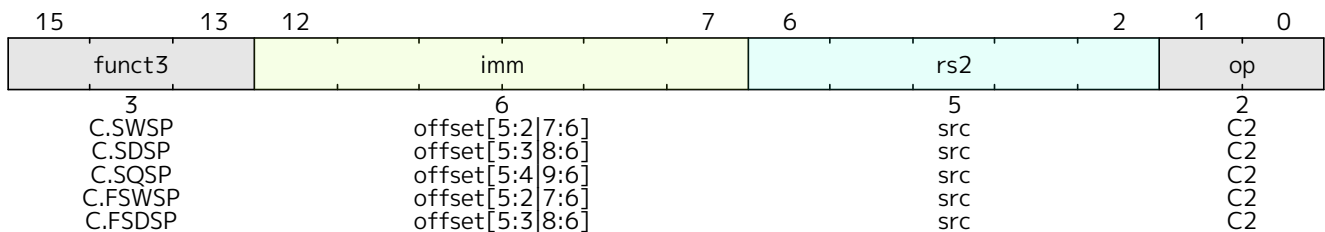
C.LWSP loads a 32-bit value from memory into register *rd*. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, *x2*. It expands to **lw rd, offset(x2)**. C.LWSP is only valid when *rd*≠*x0* the code points with *rd*=*x0* are reserved.

C.LDSP is an RV64C/RV128C-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 only valid when *rd*≠*x0* the code points with *rd*=*x0* are reserved.

C.LQSP is an RV128C-only instruction that loads a 128-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, *x2*. It expands to **lq rd, offset(x2)**. C.LQSP is only valid when *rd*≠*x0* the code points with *rd*=*x0* are reserved.

C.FLWSP is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, *x2*. It expands to **flw rd, offset(x2)**.

C.FLDSP is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, *x2*. It expands to **fld rd, offset(x2)**.



These instructions use the CSS format.

C.SWSP stores a 32-bit value in register *rs2* to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, *x2*. It expands to **sw rs2, offset(x2)**.

C.SDSP is an RV64C/RV128C-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.SQSP is an RV128C-only instruction that stores a 128-bit value in register *rs2* to memory. It computes an effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, *x2*. It expands to **sq rs2, offset(x2)**.

C.FSWSP is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, *x2*. It expands to **fsw *rs2*, offset(*x2*)**.

C.FSDSP is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, *x2*. It expands to **fsd *rs2*, offset(*x2*)**.

Register save/restore code at function entry/exit represents a significant portion of static code size. The stack-pointer-based compressed loads and stores in RVC are effective at reducing the save/restore static code size by a factor of 2 while improving performance by reducing dynamic instruction bandwidth.

A common mechanism used in other ISAs to further reduce save/restore code size is load-multiple and store-multiple instructions. We considered adopting these for RISC-V but noted the following drawbacks to these instructions:

- *These instructions complicate processor implementations.*
- *For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions.*
- *Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple.*
- *Unlike the rest of the RVC instructions, the compiler would have to be aware of these instructions to both generate the instructions and to allocate registers in an order to maximize the chances of the them being saved and stored, since they would be saved and restored in sequential order.*
- *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.



18.3.2. 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.LQ						offset[5 4 8]						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/RV128C-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.LQ is an RV128C-only instruction that loads a 128-bit value from memory into register rd' . It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register $rs1'$. It expands to **lq 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.SQ			offset[5:4 8]			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/RV128C-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.SQ is an RV128C-only instruction that stores a 128-bit value in register $rs2'$ to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register

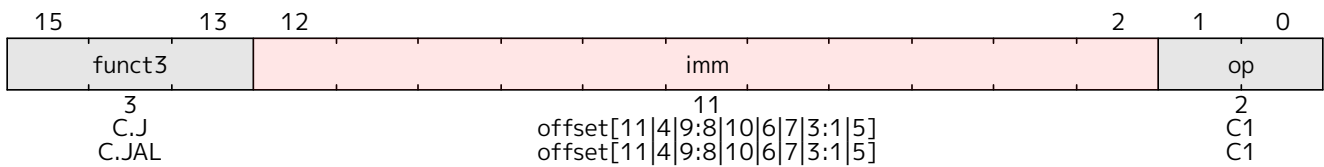
$rs1'$. It expands to **sq $rs2'$, offset($rs1'$)**.

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($rs1'$)**.

18.4. Control Transfer Instructions

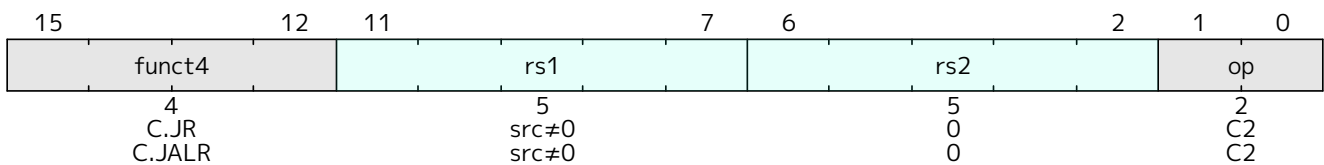
RVC provides unconditional jump instructions and conditional branch instructions. As with base RVI instructions, the offsets of all RVC control transfer instructions are in multiples of 2 bytes.



These instructions use the CJ format.

C.J performs an unconditional control transfer. The offset is sign-extended and added to the **pc** to form the jump target address. C.J can therefore target a ± 2 KiB range. C.J expands to **jal $x0$, offset**.

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



These instructions use the CR format.

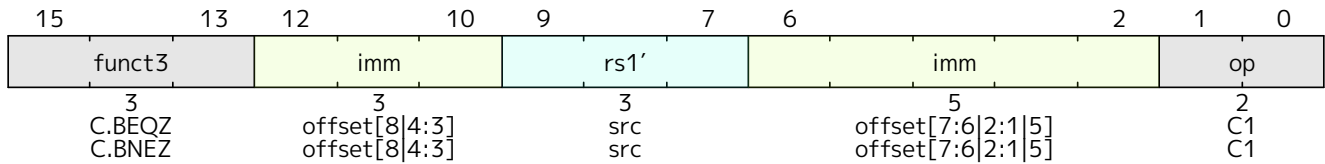
C.JR (jump register) performs an unconditional control transfer to the address in register $rs1$. C.JR expands to **jalr $x0$, 0($rs1$)**. C.JR is only valid when $rs1 \neq x0$; the code point with $rs1 = x0$ is reserved.

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (**pc+2**) to the link register, **x1**. C.JALR expands to **jalr $x1$, 0($rs1$)**. C.JALR is only valid 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**.

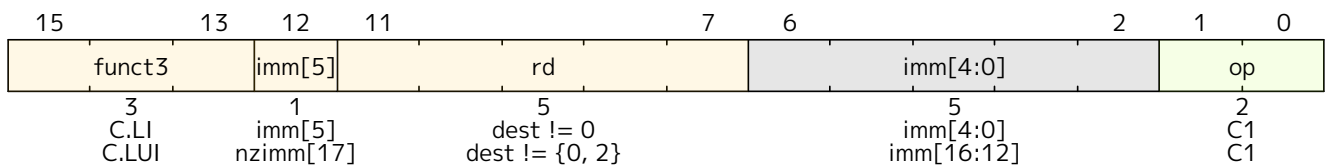
C.BNEZ is defined analogously, but it takes the branch if *rs1'* contains a nonzero value. It expands to **bne rs1', x0, offset**.

18.5. Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

18.5.1. Integer Constant-Generation Instructions

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

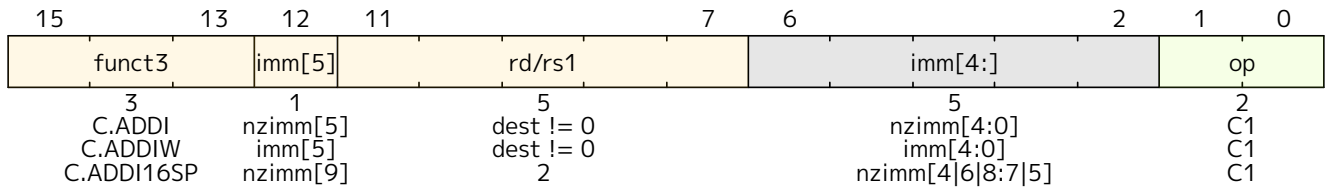


C.LI loads the sign-extended 6-bit immediate, *imm*, into register *rd*. C.LI expands into **addi rd, x0, imm**. C.LI is only valid when *rd* \neq **x0**; the code points with *rd*=**x0** encode HINTs.

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 only valid when *rd* \neq {**x0**, **x2**}, and when the immediate is not equal to zero. The code points with *imm*=0 are reserved; the remaining code points with *rd*=**x0** are HINTs; and the remaining code points with *rd*=**x2** correspond to the C.ADDI16SP instruction.

18.5.2. 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.



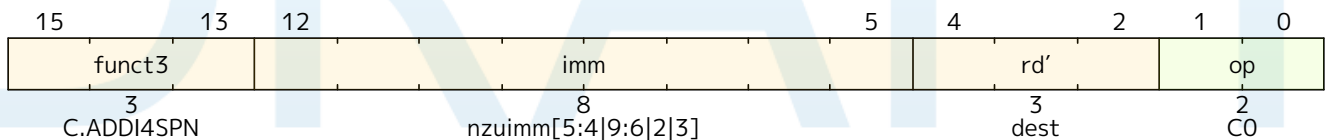
C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register *rd* then writes the result to *rd*. C.ADDI expands into **addi rd, rd, imm**. C.ADDI is only valid when *rd*≠*x0* and *imm*≠0. The code points with *rd*=*x0* encode the C.NOP instruction; the remaining code points with *imm*=0 encode HINTs.

C.ADDIW is an RV64C/RV128C-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 only valid when *rd*≠*x0*; the code points with *rd*=*x0* are reserved.

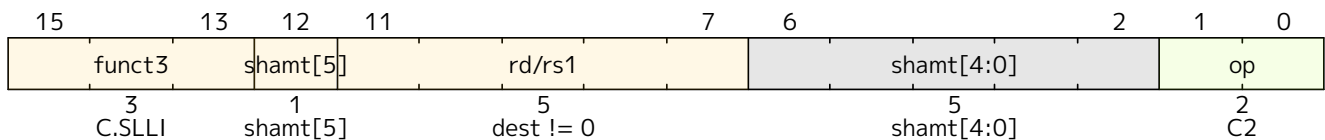
C.ADDI16SP shares the opcode with C.LUI, but has a destination field of *x2*. C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (*sp*=*x2*), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into **addi x2, x2, nzimm[9:4]**. C.ADDI16SP is only valid when *nzimm*≠0; the code point with *nzimm*=0 is reserved.



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

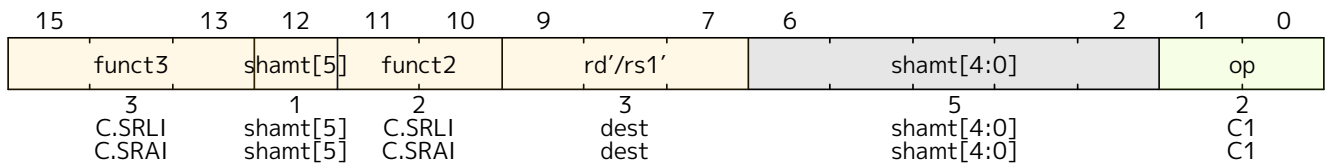


C.ADDI4SPN is a CIW-format instruction that adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, *x2*, and writes the result to *rd'*. This instruction is used to generate pointers to stack-allocated variables, and expands to **addi rd', x2, nzuimm[9:2]**. C.ADDI4SPN is only valid when *nzuimm*≠0; the code points with *nzuimm*=0 are reserved.



C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field. For RV128C, a shift amount of zero is used to encode a shift of 64. C.SLLI expands into **slli rd, rd, shamt[5:0]**, except for RV128C with *shamt*=0, which expands to **slli rd, rd, 64**.

For RV32C, *shamt*[5] must be zero; the code points with *shamt*[5]=1 are designated for custom extensions. For RV32C and RV64C, the shift amount must be non-zero; the code points with *shamt*=0 are HINTs. For all base ISAs, the code points with *rd*=*x0* are HINTs, except those with *shamt*[5]=1 in RV32C.



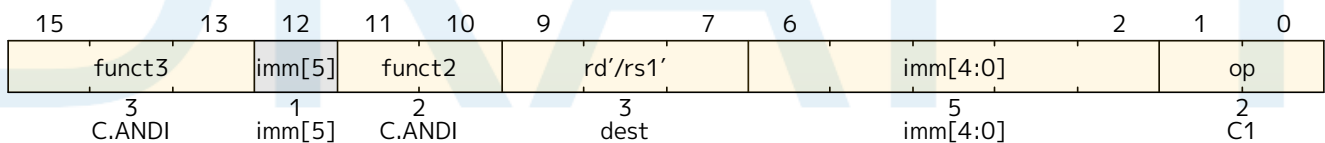
C.SRLI is a CB-format instruction that performs a logical right shift of the value in register rd' then writes the result to rd' . The shift amount is encoded in the $shamt$ field. For RV128C, a shift amount of zero is used to encode a shift of 64. Furthermore, the shift amount is sign-extended for RV128C, and so the legal shift amounts are 1-31, 64, and 96-127. C.SRLI expands into `srli rd', rd', shamt`, except for RV128C with $shamt=0$, which expands to `srli rd', rd', 64`.

For RV32C, $shamt[5]$ must be zero; the code points with $shamt[5]=1$ are designated for custom extensions. For RV32C and RV64C, the shift amount must be non-zero; the code points with $shamt=0$ are HINTs.

C.SRAI is defined analogously to C.SRLI, but instead performs an arithmetic right shift. C.SRAI expands to `srai rd', rd', shamt`.

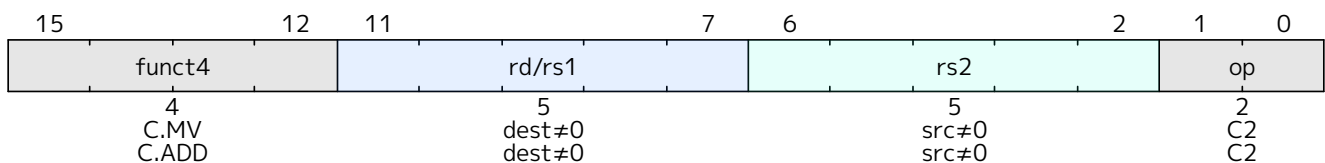


Left shifts are usually more frequent than right shifts, as left shifts are frequently used to scale address values. Right shifts have therefore been granted less encoding space and are placed in an encoding quadrant where all other immediates are sign-extended. For RV128, the decision was made to have the 6-bit shift-amount immediate also be sign-extended. Apart from reducing the decode complexity, we believe right-shift amounts of 96-127 will be more useful than 64-95, to allow extraction of tags located in the high portions of 128-bit address pointers. We note that RV128C will not be frozen at the same point as RV32C and RV64C, to allow evaluation of typical usage of 128-bit address-space codes.



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`.

18.5.3. 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 only valid when $rs2 \neq x0$ the code points with $rs2=x0$ correspond to the C.JR instruction. The code points with $rs2 \neq x0$ 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.

15	10	9	7	6	5	4	2	1	0				
funct6						rd'/rs1'		funct2		rs2'		op	
6						3		2		3		2	
C.AND						dest		C.AND		src		C1	
C.OR						dest		C.OR		src		C1	
C.XOR						dest		C.XOR		src		C1	
C.SUB						dest		C.SUB		src		C1	
C.ADDW						dest		C.ADDW		src		C1	
C.SUBW						dest		C.SUBW		src		C1	

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/RV128C-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/RV128C-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.

18.5.4. Defined Illegal Instruction

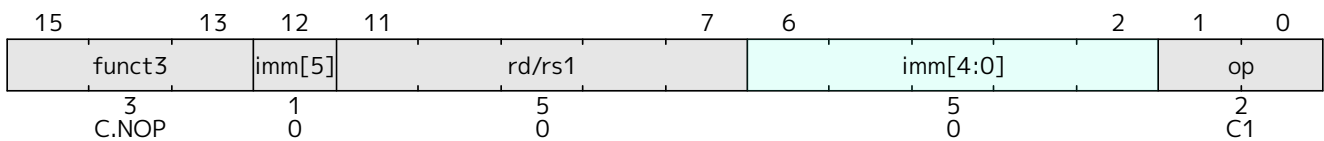
15	13	12	11	7	6	2	1	0
0	0	0	0	0	0	0	0	0
3	1	5	5	2	0	0	0	0

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



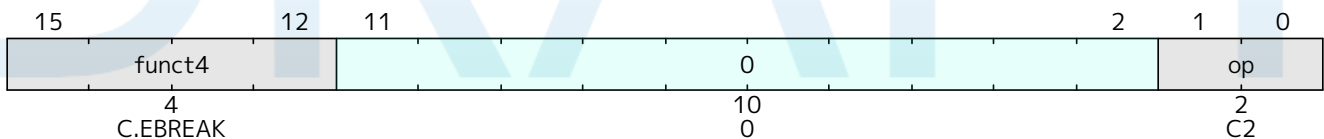
We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zero-ed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very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

18.5.5. 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**. **C.NOP** is only valid when *imm*=0; the code points with *imm*≠0 encode HINTs.

18.5.6. 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.

18.6. 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 10.3](#), 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.

18.7. 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 32](#) 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 32. RVC HINT instructions.

Instruction	Constraints	Code Points	Purpose
C.NOP	$imm \neq 0$	63	Designated for future standard use
C.ADDI	$rd \neq x0, imm = 0$	31	
C.LI	$rd = x0$	64	
C.LUI	$rd = x0, imm \neq 0$	63	
C.MV	$rd = x0, rs2 \neq x0$	31	
C.ADD	$rd = x0, rs2 \neq x0, rs2 \neq x2-x5$	27	
C.ADD	$rd = x0, rs2 \neq x2-x5$	4	($rs2=x2$) C.NTL.P1 ($rs2=x3$) C.NTL.PALL ($rs2=x4$) C.NTL.S1 ($rs2=x5$) C.NTL.ALL

Instruction	Constraints	Code Points	Purpose
C.SLLI	$rd \neq x0, imm \neq 0$	31 (RV32), 63 (RV64/128)	<i>Designated for custom use</i>
C.SLLI64	$rd = x0$	1	
C.SLLI64	$rd \neq x0$, RV32 and RV64 only	31	
C.SRLI64	RV32 and RV64 only	8	
C.SRAI64	RV32 and RV64 only	8	

18.8. RVC Instruction Set Listings

Table 33 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 18.7).

Table 33. RVC opcode map instructions.

inst[15:13] inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD LD	Reserved	FSD FSD SQSW	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDI W ADDI W	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LDSP	LWSP P	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP P FSDSP P SQSP	SWS P	FSWSP P SDSP SDSP	RV32 RV64 RV128
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 <small>RES, uimm=0</small>			
001	uimm[5:3]				rs1'			uimm[7:6]			rd'	00	C.FLD <small>(RV32/64)</small>			
001	uimm[5:4 8]				rs1'			uimm[7:6]			rd'	00	C.LQ <small>(RV128)</small>			
010	uimm[5:3]				rs1'			uimm[2 6]			rd'	00	C.LW			
011	uimm[5:3]				rs1'			uimm[2 6]			rd'	00	C.FLW <small>(RV32)</small>			
011	uimm[5:3]				rs1'			uimm[7:6]			rd'	00	C.LD <small>(RV64/128)</small>			
100	---										00				Reserved	
101	uimm[5:3]				rs1'			uimm[7:6]			rs2'	00	C.FSD <small>(RV32/64)</small>			
101	uimm[5:4 8]				rs1'			uimm[7:6]			rs2'	00	C.SQ <small>(RV128)</small>			
110	uimm[5:3]				rs1'			uimm[2 6]			rs2'	00	C.SW			
111	uimm[5:3]				rs1'			uimm[2 6]			rs2'	00	C.FSW <small>(RV32)</small>			
111	uimm[5:3]				rs1'			uimm[7:6]			rs2'	00	C.SD <small>(RV64/128)</small>			

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		<i>C.NOP</i> (<i>HINT</i> , <i>imm</i> =0)
000		imm[5]		rs1/rd≠0							imm[4:0]			01		<i>C.ADDI</i> (<i>HINT</i> , <i>imm</i> =0)
001				imm[11 4 9:8 10 6 7 3:1 5]										01		<i>C.JAL</i> (<i>RV32</i>)
001		imm[5]		rs1/rd≠0							imm[4:0]			01		<i>C.ADDIW</i> (<i>RV64/128</i> ; <i>RES</i> , <i>rd</i> =0)
010		imm[5]		rd≠0							imm[4:0]			01		<i>C.LI</i> (<i>HINT</i> , <i>rd</i> =0)
011		imm[9]		2							imm[4 6 8:7 5]			01		<i>C.ADDH16SP</i> (<i>RES</i> , <i>imm</i> =0)
011		imm[17]		rd≠{0, 2}							imm[16:12]			01		<i>C.LUI</i> (<i>RES</i> , <i>imm</i> =0; <i>HINT</i> , <i>rd</i> =0)
100		uimm[5]		00			rs1'/rd'				uimm[4:0]			01		<i>C.SRLI</i> (<i>RV32 Custom</i> , <i>uimm</i> [5]=1)
100		0		00			rs1'/rd'				0			01		<i>C.SRLI64</i> (<i>RV128</i> ; <i>RV32/64 HINT</i>)
100		uimm[5]		01			rs1'/rd'				uimm[4:0]			01		<i>C.SRAI</i> (<i>RV32 Custom</i> , <i>uimm</i> [5]=1)
100		0		01			rs1'/rd'				0			01		<i>C.SRAI64</i> (<i>RV128</i> ; <i>RV32/64 HINT</i>)
100		imm[5]		10			rs1'/rd'				imm[4:0]			01		<i>C.ANDI</i>
100		0		11			rs1'/rd'		00		rs2'			01		<i>C.SUB</i>
100		0		11			rs1'/rd'		01		rs2'			01		<i>C.XOR</i>
100		0		11			rs1'/rd'		10		rs2'			01		<i>C.OR</i>
100		0		11			rs1'/rd'		11		rs2'			01		<i>C.AND</i>
100		1		11			rs1'/rd'		00		rs2'			01		<i>C.SUBW</i> (<i>RV64/128</i> ; <i>RV32 RES</i>)
100		1		11			rs1'/rd'		01		rs2'			01		<i>C.ADDW</i> (<i>RV64/128</i> ; <i>RV32 RES</i>)
100		1		11			---		10		---			01		Reserved
100		1		11			---		11		---			01		Reserved
101				imm[11 4 9:8 10 6 7 3:1 5]										01		<i>C.J</i>
110			imm[8 4:3]				rs1'				imm[7:6 2:1 5]			01		<i>C.BEQZ</i>
111			imm[8 4:3]				rs1'				imm[7:6 2:1 5]			01		<i>C.BNEZ</i>

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			uimm[5]								uimm[4:0]				10	<i>C.SLLI</i> (<i>HINT</i> , <i>rd=0</i> ; <i>RV32 Custom</i> , <i>uimm[5]=1</i>)
000			0								0				10	<i>C.SLLI64</i> (<i>RV128</i> ; <i>RV32/64 HINT</i> ; <i>HINT</i> , <i>rd=0</i>)
001			uimm[5]								uimm[4:3 8:6]				10	<i>C.FLDSP</i> (<i>RV32/64</i>)
001			uimm[5]								uimm[4 9:6]				10	<i>C.LQSP</i> (<i>RV128</i> ; <i>RES</i> , <i>rd=0</i>)
010			uimm[5]								uimm[4:2 7:6]				10	<i>C.LWSP</i> (<i>RES</i> , <i>rd=0</i>)
011			uimm[5]								uimm[4:2 7:6]				10	<i>C.FWSP</i> (<i>RV32</i>)
011			uimm[5]								uimm[4:3 8:6]				10	<i>C.LDSP</i> (<i>RV64/128</i> ; <i>RES</i> , <i>rd=0</i>)
100			0								0				10	<i>C.JR</i> (<i>RES</i> , <i>rs1=0</i>)
100			0								rs2#0				10	<i>C.MV</i> (<i>HINT</i> , <i>rd=0</i>)
100			1								0				10	<i>C.EBREAK</i>
100			1								0				10	<i>C.JALR</i>
100			1								rs2#0				10	<i>C.ADD</i> (<i>HINT</i> , <i>rd=0</i>)
101											rs2				10	<i>C.FSDSP</i> (<i>RV32/64</i>)
101											rs2				10	<i>C.SQSP</i> (<i>RV128</i>)
110											rs2				10	<i>C.SWSP</i>
111											rs2				10	<i>C.FSWSP</i> (<i>RV32</i>)
111											rs2				10	<i>C.SDSP</i> (<i>RV64/128</i>)

Figure 5. Instruction listing for RVC, Quadrant 2

DRAFT

Chapter 19. "Zimop" May-Be-Operations Extension, Version 0.1



Warning! This stable specification may change before being accepted as a standard by RISC-V International.

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.

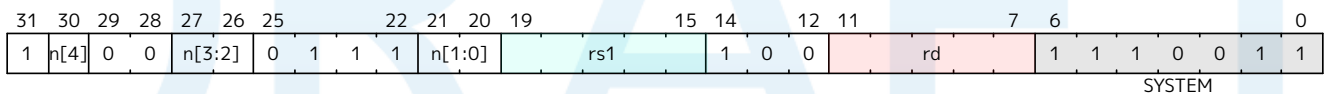


It is sometimes desirable to define instruction-set extensions whose instructions, rather than raising illegal-instruction exceptions when the extension is not implemented, take no useful action (beyond writing $x[rd]$). For example, programs with control-flow integrity checks can execute correctly on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

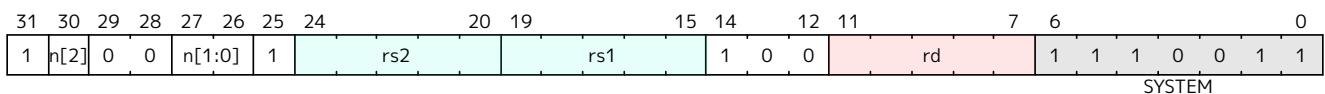
Although similar in some respects to HINTs, MOPs cannot be encoded as HINTs, because unlike HINTs, MOPs are allowed to alter architectural state.

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 $x[rs2]$.

19.1. "Zcmop" Compressed May-Be-Operations Extension, Version 0.2



Warning! This stable specification may change before being accepted as a standard by RISC-V International.

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 34. 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 requires 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	1



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 34. C.MOP. n instruction encoding.

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



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



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

Chapter 20. "B" Standard Extension for Bit Manipulation, Version 0.0

This chapter is a placeholder for a future standard extension to provide bit manipulation instructions, including instructions to insert, extract, and test bit fields, and for rotations, funnel shifts, and bit and byte permutations.



Although bit manipulation instructions are very effective in some application domains, particularly when dealing with externally packed data structures, we excluded them from the base ISAs as they are not useful in all domains and can add additional complexity or instruction formats to supply all needed operands.

We anticipate the B extension will be a brownfield encoding within the base 30-bit instruction space.

DRAFT

Chapter 21. "J" Standard Extension for Dynamically Translated Languages, Version 0.0

This chapter is a placeholder for a future standard extension to support dynamically translated languages.



Many popular languages are usually implemented via dynamic translation, including Java and Javascript. These languages can benefit from additional ISA support for dynamic checks and garbage collection.

DRAFT

Chapter 22. "P" Standard Extension for Packed-SIMD Instructions, Version 0.2



Discussions at the 5th RISC-V workshop indicated a desire to drop this packed-SIMD proposal for floating-point registers in favor of standardizing on the V extension for large floating-point SIMD operations. However, there was interest in packed-SIMD fixed-point operations for use in the integer registers of small RISC-V implementations. A task group is working to define the new P extension.

DRAFT

Chapter 23. "V" Standard Extension for Vector Operations, Version 1.0

The specification is currently hosted at github.com/riscv/riscv-v-spec.



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.

DRAFT

Chapter 24. "Zfinx", "Zdinx", "Zhinx", "Zhinxmin": Standard 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 of 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 requires 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 not provided, so 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 requires 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 requires 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

In the standard privileged architecture defined in Volume II, the **mstatus** field FS is hardwired to 0 if the Zfinx extension is implemented, and FS no longer affects the trapping behavior of floating-point instructions or **fcsr** accesses.

The **mtisa** 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. "Zfa" Standard 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.

25.1. 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 37](#). 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

<i>rs1</i>	Value	Sign	Exponent	Significand
22	3	0	10000000	100...000
23	4	0	10000001	000...000
24	8	0	10000010	000...000
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.

25.2. 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.

25.3. 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.

25.4. 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} .

25.5. 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.

25.6. 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.

DRAFT

Chapter 26. "Ztso" Standard 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 17.1](#).



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 17.1.4.1](#) 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 27. Base Cache Management Operation ISA Extensions

27.1. Pseudocode for instruction semantics

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

27.2. Introduction

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

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

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

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

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

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



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

27.3. Background

This chapter provides information common to all CMO extensions.

27.3.1. 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 a physical address 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](#).

27.3.2. 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](#).

27.4. 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*.

27.4.1. Memory Ordering

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.

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.

27.4.2. 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.

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.

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 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.



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.

Address Misaligned Exceptions

CMO instructions do *not* generate address misaligned exceptions.

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`

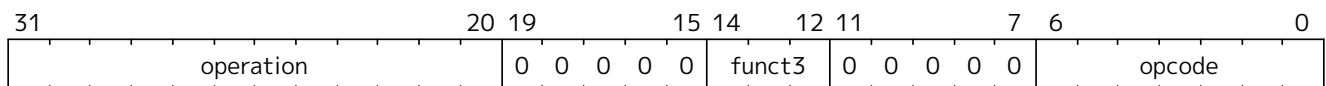
If the Zicbop extension is implemented, this specification recommends the following additional trigger module behaviors:

- 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.

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.

27.4.3. 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.

27.4.4. 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.

27.5. Control and Status Register State



The CMO extensions rely on state in `envcfg` CSRs that will be defined in a future update to the privileged architecture. If this CSR update is not ratified, the CMO extension will define its own CSRs.

Three CSRs control the execution of CMO instructions:

- `menvcfg`
- `senvcfg`
- `henvcfg`

The `senvcfg` register is used by all supervisor modes, including VS-mode. A hypervisor is responsible for saving and restoring `senvcfg` on guest context switches. The `henvcfg` register is only present if the H-extension is implemented and enabled.

Each `xenvcfg` register (where *x* is *m*, *s*, or *h*) has the following generic format:

Table 36. Generic Format for `xenvcfg` CSRs

Bits	Name	Description
[5:4]	CBIE	<p>Cache Block Invalidate instruction Enable</p> <p>Enables the execution of the cache block invalidate instruction, CBO.INVALID, in a lower privilege mode:</p> <ul style="list-style-type: none"> • 00: The instruction raises an illegal instruction or virtual instruction exception • 01: The instruction is executed and performs a flush operation • 10: <i>Reserved</i> • 11: The instruction is executed and performs an invalidate operation
[6]	CBCFE	<p>Cache Block Clean and Flush instruction Enable</p> <p>Enables the execution of the cache block clean instruction, CBO.CLEAN, and the cache block flush instruction, CBO.FLUSH, in a lower privilege mode:</p> <ul style="list-style-type: none"> • 0: The instruction raises an illegal instruction or virtual instruction exception • 1: The instruction is executed
[7]	CBZE	<p>Cache Block Zero instruction Enable</p> <p>Enables the execution of the cache block zero instruction, CBO.ZERO, in a lower privilege mode:</p> <ul style="list-style-type: none"> • 0: The instruction raises an illegal instruction or virtual instruction exception • 1: The instruction is executed

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 CBO.INVALID and perform flush operation>
    }
    else
    {
        <execute CBO.INVALID and perform invalidate operation>
    }
}

```



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

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

```

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

```

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

```

// illegal instruction exceptions
if (((priv_mode != M) && !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 CBO.ZERO>
  }

```

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

27.6. Extensions

CMO instructions are defined in the following extensions:

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

27.6.1. 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

27.6.2. 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

27.6.3. 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

27.7.2. cbo.flush

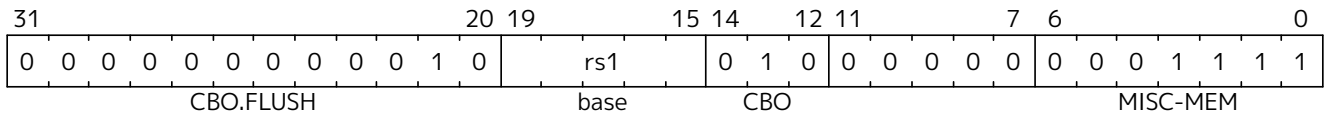
Synopsis

Perform a flush operation on a cache block

Mnemonic

cbo.flush *offset(base)*

Encoding



Description

A **cbo.flush** instruction performs a flush operation on the cache block whose 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.

Operation

TODO

27.7.3. cbo.inval

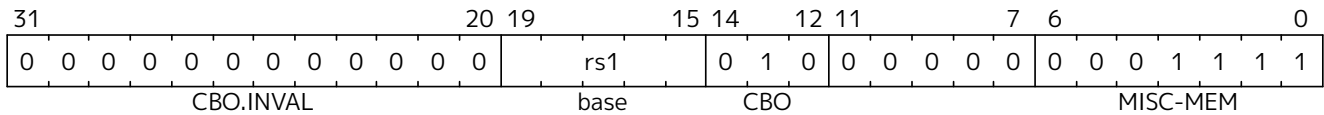
Synopsis

Perform an invalidate operation on a cache block

Mnemonic

cbo.inval *offset(base)*

Encoding



Description

A **cbo.inval** instruction performs an invalidate operation on the cache block 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. Depending on CSR programming, the instruction may perform a flush operation instead of an invalidate operation.

Operation

TODO

27.7.4. cbo.zero

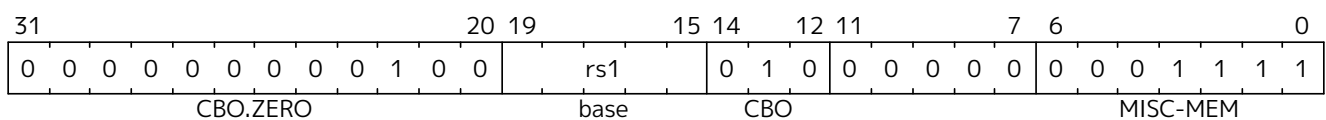
Synopsis

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

Mnemonic

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

Encoding



Description

A **cbo.zero** instruction performs stores of zeros to the full set of bytes corresponding to 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. An implementation may or may not update the entire set of bytes atomically.

Operation

TODO

27.7.5. prefetch.i

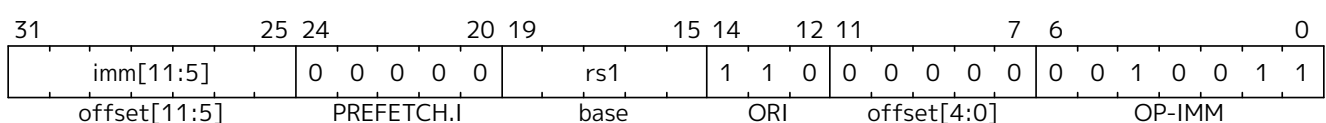
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by an instruction fetch in the near future

Mnemonic

prefetch.i *offset(base)*

Encoding



Description

A `prefetch.i` instruction indicates to hardware that the cache block whose effective address is the

sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals **0b00000**, is likely to be accessed by an instruction fetch in the near future.



An implementation may opt to cache a copy of the cache block in a cache accessed by an instruction fetch in order to improve memory access latency, but this behavior is not required.

Operation

TODO

27.7.6. prefetch.r

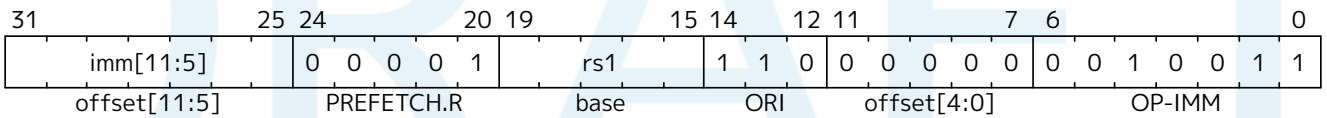
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data read in the near future

Mnemonic

prefetch.r offset(base)

Encoding



Description

A **prefetch.r** instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals **0b00000**, is likely to be accessed by a data read (i.e. load) in the near future.



An implementation may opt to cache a copy of the cache block in a cache accessed by a data read in order to improve memory access latency, but this behavior is not required.

Operation

TODO

27.7.7. prefetch.w

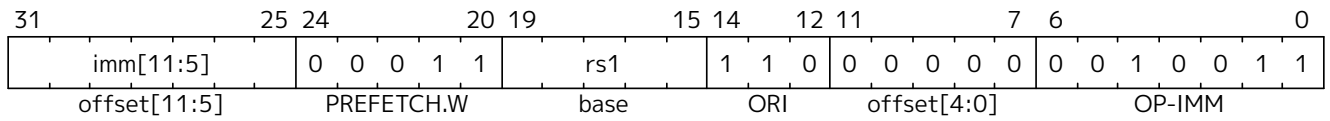
Synopsis

Provide a HINT to hardware that a cache block is likely to be accessed by a data write in the near future

Mnemonic

prefetch.w *offset(base)*

Encoding



Description

A **prefetch.w** instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in *rs1* and the sign-extended offset encoded in *imm[11:0]*, where *imm[4:0]* equals **0b00000**, is likely to be accessed by a data write (i.e. store) in the near future.



An implementation may opt to cache a copy of the cache block in a cache accessed by a data write in order to improve memory access latency, but this behavior is not required.

Operation

TODO

DRAFT

Chapter 28. "Zawrs" Statndard extension for Wait-on-Reservation-Set instructions, Version 1.01

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

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

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

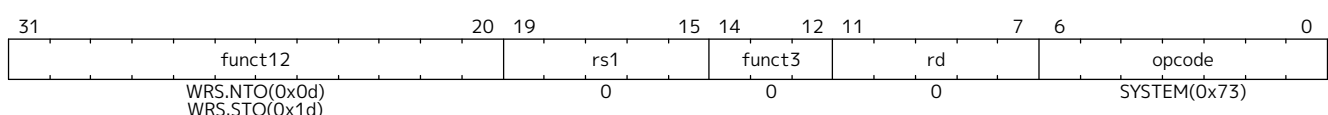
Sometimes the program waiting on a memory update may also need to carry out a task at a future time or otherwise place an upper bound on the wait. To support such use cases a second instruction **WRS.STO** (WRS-with-short-timeout) is provided that works like **WRS.NTO** but bounds the stall duration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



The instructions in the Zawrs extension are only useful in conjunction with the LR instruction, which is provided by the A extension, and which we also expect to be provided by a narrower Zalrsc extension in the future.

28.1. 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. These instructions are not supported in a constrained LR/SC loop.



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

- a. The reservation set is valid
- b. If **WRS.STO**, a "short" duration since start of stall has not elapsed
- c. No pending interrupt is observed (see the rules below)

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

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

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

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



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

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

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

Chapter 29. "Zc*" Standard Extension for Code Size Reduction

29.1. Zc* Overview

Zc* is a group of extensions that define subsets of the existing C extension (Zca, Zcd, Zcf) and new extensions which only contain 16-bit encodings.

Zcm* all reuse the encodings for *c.fld*, *c.fsd*, *c.fldsp*, *c.fsdsp*.

Table 37. 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		

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
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	
cm.popretz					yes	
cm.mva0ls					yes	
cm.mvsa0l					yes	
Table jump which overlaps with <i>c.fsdsp</i>. Complex operations intended for embedded CPUs						
cm.jt						yes
cm.jalt						yes

29.2. C

The C extension is the superset of the following extensions:

- Zca
- Zcf if F is specified (RV32 only)
- Zcd if D is specified

As C defines the same instructions as Zca, Zcf and Zcd, the rule is that:

- C always implies Zca
- C+F implies Zcf (RV32 only)
- C+D implies Zcd

29.3. Zce

The Zce extension is intended to be used for microcontrollers, and includes all relevant Zc extensions.

- Specifying Zce on RV32 without F includes Zca, Zcb, Zcmp, Zcmt
- Specifying Zce on RV32 with F includes Zca, Zcb, Zcmp, Zcmt and Zcf
- Specifying Zce on RV64 always includes Zca, Zcb, Zcmp, Zcmt
 - Zcf doesn't exist for RV64

Therefore common ISA strings can be updated as follows to include the relevant Zc extensions, for example:

- RV32IMC becomes RV32IM_Zce
- RV32IMCF becomes RV32IMF_Zce

29.4. MISA.C

MISA.C is set if the following extensions are selected:

- Zca and not F
- Zca, Zcf and F is specified (RV32 only)
- Zca, Zcf and Zcd if D is specified (RV32 only)
 - this configuration excludes Zcmp, Zcmt
- Zca, Zcd if D is specified (RV64 only)
 - this configuration excludes Zcmp, Zcmt

29.5. Zca

The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores.

Therefore it *excluded* all 16-bit floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*, *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.



the C extension only includes F/D instructions when D and F are also specified

29.6. Zcf (RV32 only)

Zcf is the existing set of compressed single precision floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*.

Zcf is only relevant to RV32, it cannot be specified for RV64.

The Zcf extension depends on the [Zca](#) and F extensions.

29.7. Zcd

Zcd is the existing set of compressed double precision floating point loads and stores: *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.

The Zcd extension depends on the [Zca](#) and D extensions.

29.8. Zcb

Zcb has simple code-size saving instructions which are easy to implement on all CPUs.

All encodings are currently reserved for all architectures, and have no conflicts with any existing extensions.



Zcb can be implemented on any CPU as the instructions are 16-bit versions of existing 32-bit instructions from the application class profile.

The Zcb extension depends on the [Zca](#) extension.

As shown on the individual instruction pages, many of the instructions in Zcb depend upon another extension being implemented. For example, *c.mul* is only implemented if M or Zmmul is implemented, and *c.sext.b* is only implemented if Zbb is implemented.

The *c.mul* encoding uses the CA register format along with other instructions such as *c.sub*, *c.xor* etc.



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

29.9. Zcmp

The Zcmp extension is a set of instructions which may be executed as a series of existing 32-bit RISC-V instructions.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with [Zcd](#), which is included when C and D extensions are both present.



Zcmp is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with architecture 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 { <i>reg_list</i> }, - <i>stack_adj</i>	cm.push
yes	yes	cm.pop { <i>reg_list</i> }, <i>stack_adj</i>	cm.pop
yes	yes	cm.popret { <i>reg_list</i> }, <i>stack_adj</i>	cm.popret
yes	yes	cm.popretz { <i>reg_list</i> }, <i>stack_adj</i>	cm.popretz
yes	yes	cm.mva01s <i>rs1'</i> , <i>rs2'</i>	Move two s0-s7 registers into a0-a1
yes	yes	cm.mvsa01 <i>r1s'</i> , <i>r2s'</i>	Move a0-a1 into two different s0-s7 registers

29.10. Zcmt

Zcmt adds the table jump instructions and also adds the JVT CSR. The JVT CSR requires a state enable if Smstateen is implemented. See [JVT CSR, table jump base vector and control register](#) for details.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with [Zcd](#), which is included when C and D extensions are both present.



Zcmt is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with RVA profiles.

The Zcmt extension depends on the [Zca](#) and Zicsr extensions.

RV32	RV64	Mnemonic	Instruction
yes	yes	cm.jt <i>index</i>	Jump via table
yes	yes	cm.jalt <i>index</i>	Jump and link via table

29.11. Zc instruction formats

Several instructions in this specification use the following new instruction formats.

Format	instructions	15:10	9	8	7	6	5	4	3	2	1	0	
CLB	c.lbu	funct6	rs1'			uimm		rd'			op		
CSB	c.sb	funct6	rs1'			uimm		rs2'			op		
CLH	c.lhu, c.lh	funct6	rs1'			func t1	uim m	rd'			op		
CSH	c.sh	funct6	rs1'			func t1	uim m	rs2'			op		
CU	c.[sz]ext.*, c.not	funct6	rd'/rs1'			funct5					op		
CMMV	cm.mvsa01 cm.mva01s	funct6	r1s'			funct2		r2s'			op		
CMJT	cm.jt cm.jalt	funct6	index									op	
CMPP	cm.push*, cm.pop*	funct6	funct2		urlist				spimm		op		



c.mul uses the existing CA format.

29.12. Zcb instructions

29.12.1. c.lbu

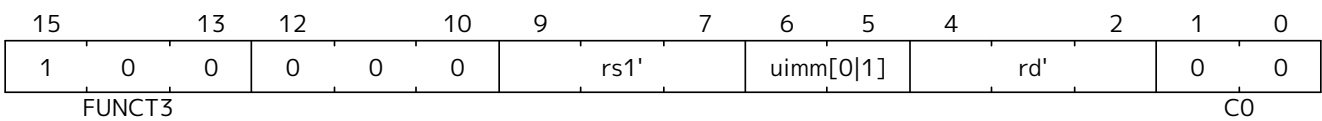
Synopsis:

Load unsigned byte, 16-bit encoding

Mnemonic:

`c.lbu rd', uimm(rs1')`

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]     = encoding[5];
uimm[0]     = encoding[6];
```

Description:

This instruction loads a byte from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting byte is zero extended to XLEN bits and is written to *rd'*.

 *rd' and rs1' are from the standard 8-register set x8-x15.*

Prerequisites:

None

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

X(rdc) = EXTZ(mem[X(rs1c)+EXTZ(uimm)][7..0]);
```

29.12.2. c.lhu

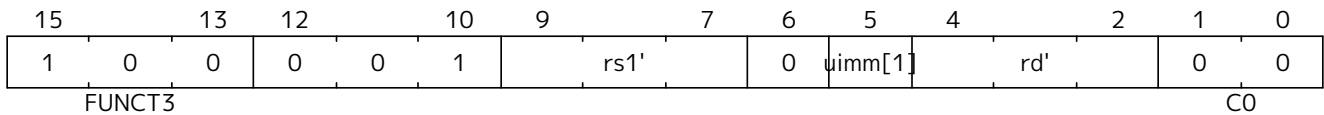
Synopsis:

Load unsigned halfword, 16-bit encoding

Mnemonic:

c.lhu rd' , $uimm(rs1')$

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description:

This instruction loads a halfword from the memory address formed by adding $rs1'$ to the zero extended immediate $uimm$. The resulting halfword is zero extended to XLEN bits and is written to rd' .



rd' and $rs1'$ are from the standard 8-register set $x8-x15$.

Prerequisites:

None

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

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

29.12.3. c.lh

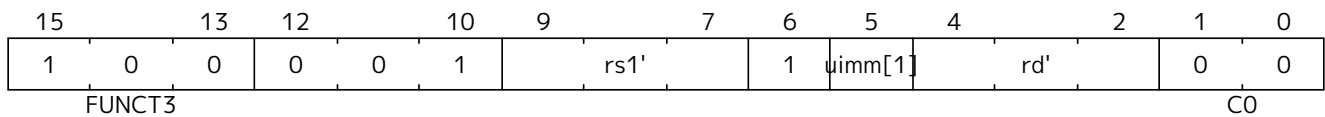
Synopsis:

Load signed halfword, 16-bit encoding

Mnemonic:

c.lh *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]     = encoding[5];
uimm[0]     = 0;
```

Description:

This instruction loads a halfword from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting halfword is sign extended to XLEN bits and is written to *rd'*.



rd' and *rs1'* are from the standard 8-register set *x8-x15*.

Prerequisites:

None

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

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

29.12.4. c.sb

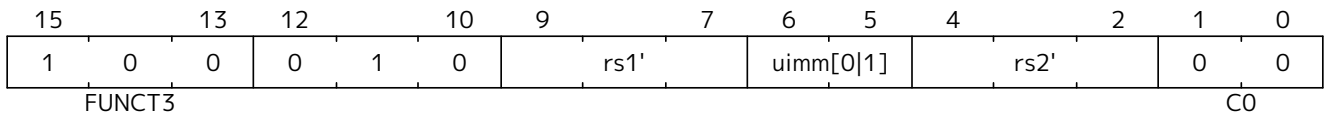
Synopsis:

Store byte, 16-bit encoding

Mnemonic:

`c.sb rs2', uimm(rs1')`

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]     = encoding[5];
uimm[0]     = encoding[6];
```

Description:

This instruction stores the least significant byte of `rs2'` to the memory address formed by adding `rs1'` to the zero extended immediate `uimm`.



rs1' and rs2' are from the standard 8-register set x8-x15.

Prerequisites:

None

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
mem[X(rs1c)+EXTZ(uimm)][7..0] = X(rs2c)
```

29.12.5. c.sh

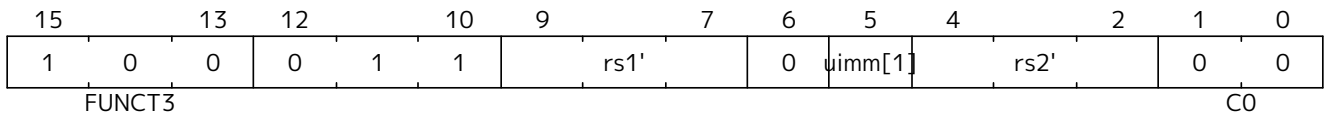
Synopsis:

Store halfword, 16-bit encoding

Mnemonic:

c.sh *rs2'*, *uimm(rs1')*

Encoding (RV32, RV64):



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]     = encoding[5];
uimm[0]     = 0;
```

Description:

This instruction stores the least significant halfword of *rs2'* to the memory address formed by adding *rs1'* to the zero extended immediate *uimm*.



rs1' and *rs2'* are from the standard 8-register set x8-x15.

Prerequisites:

None

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][15..0] = X(rs2c)
```

29.12.6. c.zext.b

Synopsis:

Zero extend byte, 16-bit encoding

Mnemonic:

c.zext.b *rd'/rs1'*

Encoding (RV32, RV64):

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

Description:

This instruction takes a single source/destination operand. It zero-extends the least-significant byte of the operand to XLEN bits by inserting zeros into all of the bits more significant than 7.



rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites:

None

32-bit equivalent:

```
andi rd'/rs1', rd'/rs1', 0xff
```



The SAIL module variable for rd'/rs1' is called rsdc.

Operation:

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


29.12.7. c.sext.b

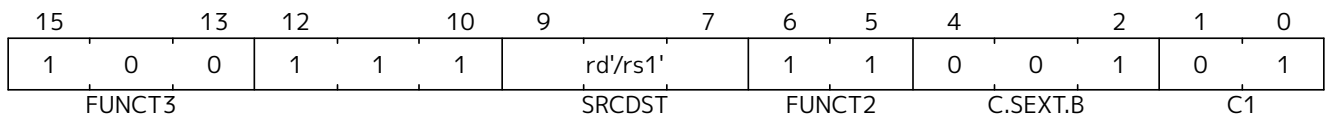
Synopsis:

Sign extend byte, 16-bit encoding

Mnemonic:

c.sext.b rd'/rs1'

Encoding (RV32, RV64):



Description:

This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand to XLEN bits by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.



rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites:

Zbb is also required.



The SAIL module variable for rd'/rs1' is called rsdc.

Operation:

X(rsdc) = EXTS(X(rsdc)[7..0]);

Prerequisites:

Zbb is also required.



The SAIL module variable for rd'/rs1' is called rsdc.

Operation:

X(rsdc) = EXTZ(X(rsdc)[15..0]);

29.12.8. c.zext.h

Synopsis:

Zero extend halfword, 16-bit encoding

Mnemonic:

`c.zext.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	0	0	1
FUNCT3			SRCDST			FUNCT2		C.ZEXT.H			C1			

Description:

This instruction takes a single source/destination operand. It zero-extends the least-significant halfword of the operand to XLEN bits by inserting zeros into all of the bits more significant than 15.



rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites:

Zbb is also required.



The SAIL module variable for rd'/rs1' is called rsdc.

Operation:

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

29.12.9. c.sext.h

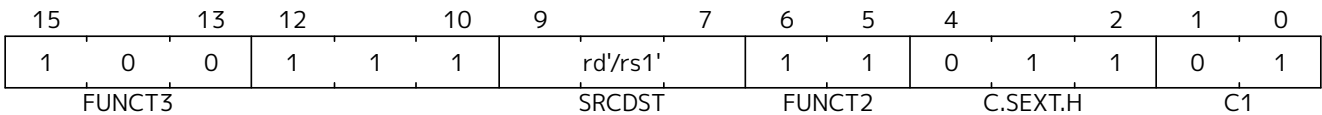
Synopsis:

Sign extend halfword, 16-bit encoding

Mnemonic:

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

Encoding (RV32, RV64):



Description:

This instruction takes a single source/destination operand. It sign-extends the least-significant halfword in the operand to XLEN bits by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.



rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites:

Zbb is also required.



*The SAIL module variable for *rd'/rs1'* is called *rsdc*.*

Operation:

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

29.12.10. c.zext.w

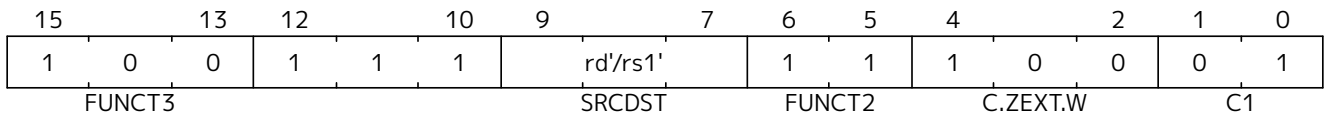
Synopsis:

Zero extend word, 16-bit encoding

Mnemonic:

`c.zext.w rd'/rs1'`

Encoding (RV64):



Description:

This instruction takes a single source/destination operand. It zero-extends the least-significant word of the operand to XLEN bits by inserting zeros into all of the bits more significant than 31.



rd'/rs1' is from the standard 8-register set x8-x15.

Prerequisites:

Zba is also required.

32-bit equivalent:

```
add.uw rd'/rs1', rd'/rs1', zero
```



The SAIL module variable for rd'/rs1' is called rsdc.

Operation:

```
X(rsdc) = EXTZ(X(rsdc)[31..0]);
```

29.12.11. c.not

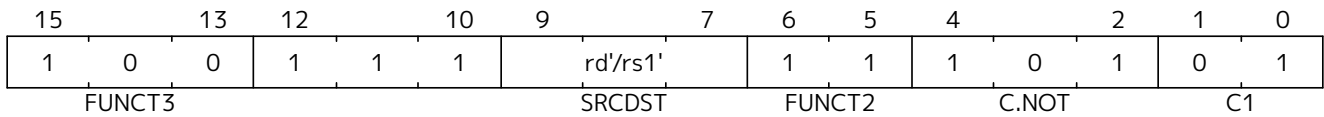
Synopsis:

Bitwise not, 16-bit encoding

Mnemonic:

c.not *rd'/rs1'*

Encoding (RV32, RV64):



Description:

This instruction takes the one’s complement of *rd'/rs1'* and writes the result to the same register.

 *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites:

None

32-bit equivalent:

```
xori rd'/rs1', rd'/rs1', -1
```

 The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation:

$$X(rsdc) = X(rsdc) \text{ XOR } -1;$$

29.12.12. c.mul

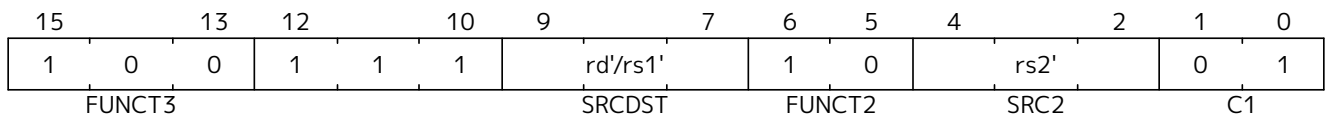
Synopsis:

Multiply, 16-bit encoding

Mnemonic:

c.mul *rsd'*, *rs2'*

Encoding (RV32, RV64):



Description:

This instruction multiplies XLEN bits of the source operands from *rsd'* and *rs2'* and writes the lowest XLEN bits of the result to *rsd'*.



rd'/rs1' and *rs2'* are from the standard 8-register set x8-x15.

Prerequisites:

M or Zmmul must be configured.



The SAIL module variable for *rd'/rs1'* is called *rsdc*, and for *rs2'* is called *rs2c*.

Operation:

```
let result_wide = to_bits(2 * sizeof(xlen), signed(X(rsdc)) * signed(X(rs2c)));
X(rsdc) = result_wide[(sizeof(xlen) - 1) .. 0];
```

29.13. PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- [cm.push](#)
- [cm.pop](#)
- [cm.popret](#)
- [cm.popretz](#)

The term PUSH refers to *cm.push*.

The term POP refers to *cm.pop*.

The term POPRET refers to *cm.popret* and *cm.popretz*.

Common details for these instructions are in this section.

29.13.1. PUSH/POP functional overview

PUSH, POP, POPRET are used to reduce the size of function prologues and epilogues.

1. The PUSH instruction
 - adjusts the stack pointer to create the stack frame
 - pushes (stores) the registers specified in the register list to the stack frame
2. The POP instruction
 - pops (loads) the registers in the register list from the stack frame
 - adjusts the stack pointer to destroy the stack frame
3. The POPRET instructions
 - pop (load) the registers in the register list from the stack frame
 - *cm.popretz* also moves zero into *a0* as the return value
 - adjust the stack pointer to destroy the stack frame
 - execute a *ret* instruction to return from the function

29.13.2. Example usage

This example gives an illustration of the use of PUSH and POPRET.

The function *processMarkers* in the EMBench benchmark picojpeg in the following file on github: [libpicojpeg.c](#)

The prologue and epilogue compile with GCC10 to:

```

0001098a <processMarkers>:
1098a:    711d                addi    sp,sp,-96 ;#cm.push(1)
1098c:    c8ca                sw      s2,80(sp) ;#cm.push(2)
1098e:    c6ce                sw      s3,76(sp) ;#cm.push(3)
10990:    c4d2                sw      s4,72(sp) ;#cm.push(4)
10992:    ce86                sw      ra,92(sp) ;#cm.push(5)
10994:    cca2                sw      s0,88(sp) ;#cm.push(6)
10996:    caa6                sw      s1,84(sp) ;#cm.push(7)
10998:    c2d6                sw      s5,68(sp) ;#cm.push(8)
1099a:    c0da                sw      s6,64(sp) ;#cm.push(9)
1099c:    de5e                sw      s7,60(sp) ;#cm.push(10)
1099e:    dc62                sw      s8,56(sp) ;#cm.push(11)
109a0:    da66                sw      s9,52(sp) ;#cm.push(12)
109a2:    d86a                sw      s10,48(sp);#cm.push(13)
109a4:    d66e                sw      s11,44(sp);#cm.push(14)
...
109f4:    4501                li      a0,0      ;#cm.popretz(1)
109f6:    40f6                lw      ra,92(sp) ;#cm.popretz(2)
109f8:    4466                lw      s0,88(sp) ;#cm.popretz(3)
109fa:    44d6                lw      s1,84(sp) ;#cm.popretz(4)
109fc:    4946                lw      s2,80(sp) ;#cm.popretz(5)
109fe:    49b6                lw      s3,76(sp) ;#cm.popretz(6)
10a00:    4a26                lw      s4,72(sp) ;#cm.popretz(7)
10a02:    4a96                lw      s5,68(sp) ;#cm.popretz(8)
10a04:    4b06                lw      s6,64(sp) ;#cm.popretz(9)
10a06:    5bf2                lw      s7,60(sp) ;#cm.popretz(10)
10a08:    5c62                lw      s8,56(sp) ;#cm.popretz(11)
10a0a:    5cd2                lw      s9,52(sp) ;#cm.popretz(12)
10a0c:    5d42                lw      s10,48(sp);#cm.popretz(13)
10a0e:    5db2                lw      s11,44(sp);#cm.popretz(14)
10a10:    6125                addi    sp,sp,96  ;#cm.popretz(15)
10a12:    8082                ret                                ;#cm.popretz(16)

```

with the GCC option `-msave-restore` the output is the following:

```
0001080e <processMarkers>:
1080e:      73a012ef          jal    t0,11f48 <__riscv_save_12>
10812:      1101             addi   sp,sp,-32
...
10862:      4501             li     a0,0
10864:      6105             addi   sp,sp,32
10866:      71e0106f         j      11f84 <__riscv_restore_12>
```

with PUSH/POPRET this reduces to

```
0001080e <processMarkers>:
1080e:      b8fa             cm.push {ra,s0-s11},-96
...
10866:      bcfa             cm.popretz {ra,s0-s11}, 96
```

The prologue / epilogue reduce from 60-bytes in the original code, to 14-bytes with `-msave-restore`, and to 4-bytes with PUSH and POPRET. As well as reducing the code-size PUSH and POPRET eliminate the branches from calling the millicode *save/restore* routines and so may also perform better.



The calls to `<riscv_save_0>/<riscv_restore_0>` become 64-bit when the target functions are out of the $\pm 1\text{MB}$ 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.

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.

Register list handling

There is no support for the `{ra, s0-s10}` register list without also adding `s11`. Therefore the `{ra, s0-s11}` register list must be used in this case.

29.13.3. PUSH/POP Fault handling

Correct execution requires that *sp* refers to idempotent memory (also see [Non-idempotent memory handling](#)), because the core must be able to handle traps detected during the sequence. The entire PUSH/POP sequence is re-executed after returning from the trap handler, and multiple traps are possible during the sequence.

If a trap occurs during the sequence then *xEPC* is updated with the PC of the instruction, *xTVAL* (if not read-only-zero) updated with the bad address if it was an access fault and *xCAUSE* updated with the type of trap.



It is implementation defined whether interrupts can also be taken during the sequence execution.

29.13.4. Software view of execution

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 pseudo-code
 - 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
```

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 pseudo-code.
 - The bytes may be loaded in any order.
 - The bytes may be grouped into larger accesses.
 - Any of the bytes may be loaded multiple times.
- A stack pointer adjustment
- An optional `li a0, 0`
- An optional `ret`

If a trap occurs during the sequence, then any loads which were executed before the trap may update architectural state. The loads will be re-executed once the trap handler completes, so the values will be overwritten. Therefore it is permitted for an implementation to update some of the destination registers before taking a fault.

The optional `li a0, 0`, stack pointer adjustment and optional `ret` must only be committed only when it is certain that the entire POP/POPRET instruction will commit.

For POPRET once the stack pointer adjustment has been committed the `ret` must execute.

For example:

```
cm.popretz {ra, s0-s3}, 32;
```

Appears to software as:

```
# any or all of these load instructions may execute multiple times
# therefore these updates may be visible in the interrupt/exception handler
lw  s3, 28(sp)
lw  s2, 24(sp)
lw  s1, 20(sp)
lw  s0, 16(sp)
lw  ra, 12(sp)

# these must only execute once, will only execute after all loads
# complete successfully all instructions must execute atomically
# therefore these updates are not visible in the interrupt/exception handler
li a0, 0
addi sp, sp, 32
ret
```

29.13.5. Non-idempotent memory handling

An implementation may have a requirement to issue a PUSH/POP instruction to non-idempotent memory.

If the core implementation does not support PUSH/POP to non-idempotent memories, the core may use an idempotency PMA to detect it and take a load (POP/POPRET) or store (PUSH) access fault exception in order to avoid unpredictable results.

Software should only use these instructions on non-idempotent memory regions when software can tolerate the required memory accesses being issued repeatedly in the case that they cause exceptions.

29.13.6. Example RV32I PUSH/POP sequences

The examples are included show the load/store series expansion and the stack adjustment. Examples of *cm.popret* and *cm.popretz* are not included, as the difference in the expanded sequence from *cm.pop* is trivial in all cases.

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;
```

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;
```

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

expands to:

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

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;
```

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;
```


29.13.7. cm.push

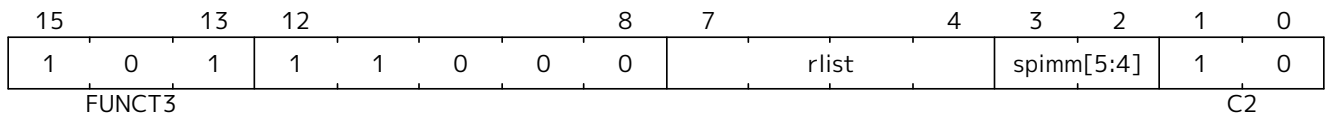
Synopsis:

Create stack frame: store ra and 0 to 12 saved registers to the stack frame, optionally allocate additional stack space.

Mnemonic:

`cm.push {reg_list}, -stack_adj`

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.push.e

Assembly Syntax:

```
cm.push {reg_list}, -stack_adj
cm.push {xreg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

RV32E:

```
switch (rlist){
  case 4: {reg_list="ra";      xreg_list="x1";}
  case 5: {reg_list="ra, s0";  xreg_list="x1, x8";}
  case 6: {reg_list="ra, s0-s1"; xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 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[5:4] * 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 [PUSH/POP Register Instructions](#).

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists

Operation:

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. 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 pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
sp-=stack_adj;
```

DRAFT

29.13.8. cm.pop

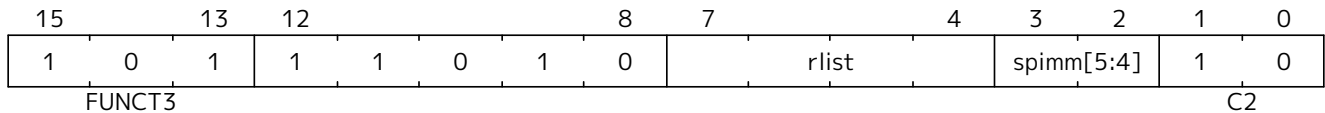
Synopsis:

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame.

Mnemonic:

`cm.pop {reg_list}, stack_adj`

Encoding (RV32, RV64):



rlist values 0 to 3 are reserved for a future EABI variant called cm.pop.e

Assembly Syntax:

```
cm.pop {reg_list}, stack_adj
cm.pop {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

RV32E:

```
switch (rlist){
  case 4: {reg_list="ra";      xreg_list="x1";}
  case 5: {reg_list="ra, s0";  xreg_list="x1, x8";}
  case 6: {reg_list="ra, s0-s1"; xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 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[5:4] * 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 [PUSH/POP Register Instructions](#).

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists

Operation:

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. 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 pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;
```

DRAFT

29.13.9. cm.popretz

Synopsis:

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, move zero into a0, return to ra.

Mnemonic:

cm.popretz {reg_list}, stack_adj

Encoding (RV32, RV64):

15	13	12		8	7		4	3	2	1	0		
1	0	1	1	1	1	0	0	rlist		spimm[5:4]		1	0
FUNCT3								C2					



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[5:4] * 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[5:4] * 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 a0 and then returns to *ra*.



All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see [PUSH/POP Register Instructions](#).

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists

Operation:

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. 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 pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.



*The li a0, 0 **could** be executed more than once, but is included in the atomic section for*

convenience.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
asm("li a0, 0");  
sp+=stack_adj;  
asm("ret");
```

DRAFT

29.13.10. cm.popret

Synopsis:

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, return to ra.

Mnemonic:

`cm.popret {reg_list}, stack_adj`

Encoding (RV32, RV64):

15	13	12				8	7			4	3	2	1	0
1	0	1	1	1	1	1	0		rlist			spimm[5:4]	1	0
FUNCT3													C2	



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[5:4] * 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[5:4] * 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 [PUSH/POP Register Instructions](#).

Stack Adjustment Calculation:

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists

Operation:

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. 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 pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;  
asm("ret");
```

DRAFT

29.13.11. cm.mvsa01

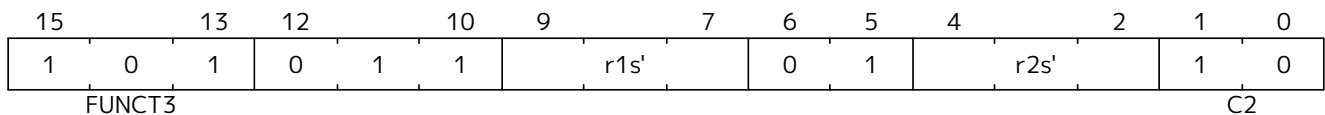
Synopsis:

Move a0-a1 into two registers of s0-s7

Mnemonic:

cm.mvsa01 *r1s'*, *r2s'*

Encoding (RV32, RV64):



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

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 pseudo-code 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 pseudo-code. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
    reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

29.13.12. cm.mva01s

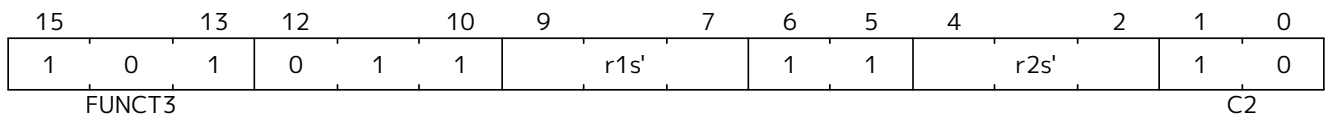
Synopsis:

Move two s0-s7 registers into a0-a1

Mnemonic:

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

Encoding (RV32, RV64):



Assembly Syntax:

```
cm.mva01s r1s', r2s'
```

Description: This instruction moves *r1s'* into *a0* and *r2s'* into *a1*. The execution is atomic, so it is not possible to observe state where only one of *a0* or *a1* have been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudo-code 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 pseudo-code. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
    reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

29.14. Table Jump Overview

cm.jt ([Jump via table](#)) and *cm.jalt* ([Jump and link via table](#)) are referred to as table jump.

Table jump uses a 256-entry XLEN wide table in instruction memory to contain function addresses. The table must be a minimum of 64-byte aligned.

Table entries follow the current data endianness. This is different from normal instruction fetch which is always little-endian.

cm.jt and *cm.jalt* encodings index the table, giving access to functions within the full XLEN wide address space.

This is used as a form of dictionary compression to reduce the code size of *jal* / *auipc+jalr* / *jr* / *auipc+jr* instructions.

Table jump allows the linker to replace the following instruction sequences with a *cm.jt* or *cm.jalt* encoding, and an entry in the table:

- 32-bit *j* calls
- 32-bit *jal* ra calls
- 64-bit *auipc+jr* calls to fixed locations
- 64-bit *auipc+jalr* ra calls to fixed locations
 - The *auipc+jr/jalr* sequence is used because the offset from the PC is out of the $\pm 1\text{MB}$ range.

If a return address stack is implemented, then as *cm.jalt* is equivalent to *jal ra*, it pushes to the stack.

29.14.1. JVT

The base of the table is in the JVT CSR (see [JVT CSR, table jump base vector and control register](#)), each table entry is XLEN bits.

If the same function is called with and without linking then it must have two entries in the table. This is typically caused by the same function being called with and without tail calling.

29.14.2. Table Jump Fault handling

For a table jump instruction, the table entry that the instruction selects is considered an extension of the instruction itself. Hence, the execution of a table jump instruction involves two instruction fetches, the first to read the instruction (*cm.jt/cm.jalt*) and the second to read from the jump vector table (JVT). Both instruction fetches are *implicit* reads, and both require execute permission; read permission is irrelevant. It is recommended that the second fetch be ignored for hardware triggers and breakpoints.

Memory writes to the jump vector table require an instruction barrier (*fence.i*) to guarantee that they are visible to the instruction fetch.

Multiple contexts may have different jump vector tables. JVT may be switched between them without an instruction barrier if the tables have not been updated in memory since the last *fence.i*.

If an exception occurs on either instruction fetch, xEPC is set to the PC of the table jump instruction, xCAUSE is set as expected for the type of fault and xTVAL (if not set to zero) contains the fetch address which caused the fault.

DRAFT

29.14.3. JVT CSR

Synopsis:

Table jump base vector and control register

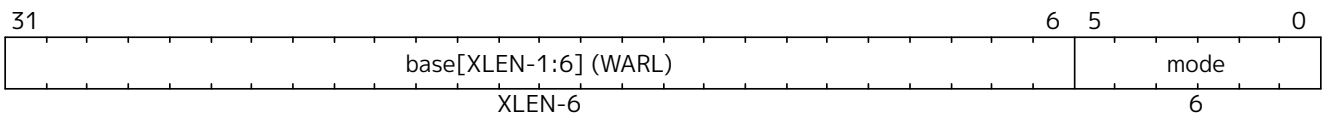
Address:

0x0017

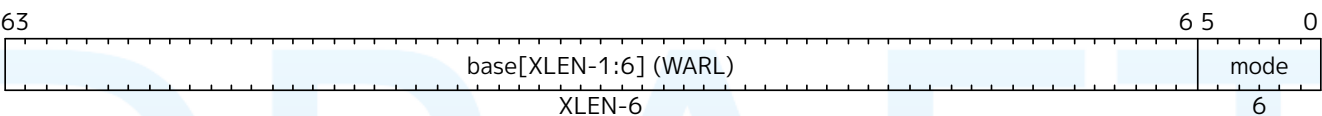
Permissions:

URW

Format (RV32):



Format (RV64):



Description:

The *JVT* register is an XLEN-bit **WARL** read/write register that holds the jump table configuration, consisting of the jump table base address (BASE) and the jump table mode (MODE).

If [Section 29.10](#) is implemented then *JVT* must also be implemented, but can contain a read-only value. If *JVT* is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 64-byte boundary.

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 38. *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 adds architectural state to the system software context (such as an OS process), therefore must be saved/restored on context switches.

State Enable:

If the *Smstateen* extension is implemented, then bit 2 in *mstateen0*, *sstateen0*, and *hstateen0* is implemented. If bit 2 of a controlling *stateen0* CSR is zero, then access to the *JVT* CSR and execution of a *cm.jalt* or *cm.jt* instruction by a lower privilege level results in an Illegal Instruction trap (or, if appropriate, a Virtual Instruction trap).

DRAFT

29.14.4. cm.jt

Synopsis:

jump via table

Mnemonic:

cm.jt *index*

Encoding (RV32, RV64):



For this encoding to decode as *cm.jt*, $index < 32$, otherwise it decodes as *cm.jalt*, see [Jump and link via table](#).



If *JVT.mode* = 0 (*Jump Table Mode*) then *cm.jt* behaves as specified here. If *JVT.mode* is a reserved value, then *cm.jt* is also reserved. In the future other defined values of *JVT.mode* may change the behaviour of *cm.jt*.

Assembly Syntax:

```
cm.jt index
```

Description:

cm.jt reads an entry from the jump vector table in memory and jumps to the address that was read.

For further information see [Table Jump Overview](#).

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.  
  
# target_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  
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];  
  
j target_address[XLEN-1:0]&~0x1;
```

DRAFT

29.14.5. cm.jalt

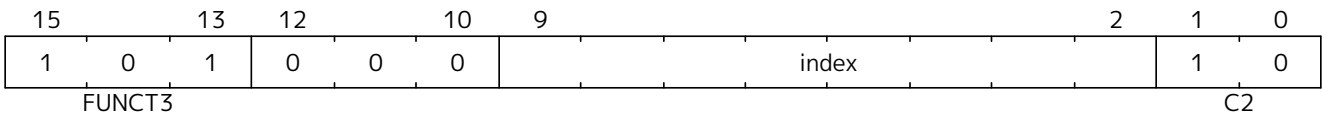
Synopsis:

jump via table with optional link

Mnemonic:

cm.jalt *index*

Encoding (RV32, RV64):



For this encoding to decode as *cm.jalt*, *index* ≥ 32, otherwise it decodes as *cm.jt*, see [Jump via table](#).



If *JVT.mode* = 0 (*Jump Table Mode*) then *cm.jalt* behaves as specified here. If *JVT.mode* is a reserved value, then *cm.jalt* is also reserved. In the future other defined values of *JVT.mode* may change the behaviour of *cm.jalt*.

Assembly Syntax:

```
cm.jalt index
```

Description:

cm.jalt reads an entry from the jump vector table in memory and jumps to the address that was read, linking to *ra*.

For further information see [Table Jump Overview](#).

Prerequisites:

None

32-bit equivalent:

No direct equivalent encoding exists.

Operation:

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.  
  
# target_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  
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];  
  
jal ra, target_address[XLEN-1:0]&~0x1;
```

DRAFT

Chapter 30. 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 39. RISC-V base opcode map, $inst[1:0]=11$

$inst[4:2]$	000	001	010	011	100	101	110	$111 (>32b)$
$inst[6:5]$								
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	$\geq 80b$

Table 39 shows a map of the major opcodes for RVG. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. 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* and *custom-1* will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked *custom-2/rv128* and *custom-3/rv128* are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions in RV32 and RV64.

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 18 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. Chapter 31 has a more extensive discussion of ways to add extensions to the RISC-V ISA.

31	27	26	25	24	20	19	15	14	12	11		7	6		0
funct7				rs2		rs1		funct3		rd			opcode		R-type
imm[11:0]						rs1		funct3		rd			opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]			opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]			opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20 10:1 11 19:12]										rd			opcode		J-type

RV32I Base Instruction Set

imm[31:12]				rd		0110111	LUI
imm[31:12]				rd		0010111	AUIPC
imm[20 10:1 11 19:12]				rd		1101111	JAL
imm[11:0]			rs1	000	rd		1100111 JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]		1100011	BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]		1100011	BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]		1100011	BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]		1100011	BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]		1100011	BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]		1100011	BGEU
imm[11:0]			rs1	000	rd		0000011 LB
imm[11:0]			rs1	001	rd		0000011 LH
imm[11:0]			rs1	010	rd		0000011 LW
imm[11:0]			rs1	100	rd		0000011 LBU
imm[11:0]			rs1	101	rd		0000011 LHU
imm[11:5]	rs2	rs1	000	imm[4:0]		0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]		0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]		0100011	SW
imm[11:0]			rs1	000	rd		0010011 ADDI
imm[11:0]			rs1	010	rd		0010011 SLTI
imm[11:0]			rs1	011	rd		0010011 SLTIU
imm[11:0]			rs1	100	rd		0010011 XORI
imm[11:0]			rs1	110	rd		0010011 ORI
imm[11:0]			rs1	111	rd		0010011 ANDI
0000000	shamt	rs1	001	rd		0010011	SLLI
0000000	shamt	rs1	101	rd		0010011	SRLI
0100000	shamt	rs1	101	rd		0010011	SRAI

0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
1000	0011	0011	00000	000	00000	0001111	FENCE.TSO
0000	0001	0000	00000	000	00000	0001111	PAUSE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

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]		0100011		SD
000000			shamt		rs1		001		rd		0010011		SLLI
000000			shamt		rs1		101		rd		0010011		SRLI
010000			shamt		rs1		101		rd		0010011		SRAI
imm[11:0]				rs1		000		rd		0011011		ADDIW	
0000000			shamt		rs1		001		rd		0011011		SLLIW
0000000			shamt		rs1		101		rd		0011011		SRLIW
0100000			shamt		rs1		101		rd		0011011		SRAIW
0000000			rs2		rs1		000		rd		0111011		ADDW
0100000			rs2		rs1		000		rd		0111011		SUBW
0000000			rs2		rs1		001		rd		0111011		SLLW
0000000			rs2		rs1		101		rd		0111011		SRLW
0100000			rs2		rs1		101		rd		0111011		SRAW

RV32/RV64 Zifencei Standard Extension

imm[11:0]				rs1		001		rd		0001111		FENCE.I	
-----------	--	--	--	-----	--	-----	--	----	--	---------	--	---------	--

RV32/RV64 Zicsr Standard Extension

csr				rs1		001		rd		1110011		CSRRW	
csr				rs1		010		rd		1110011		CSRRS	
csr				rs1		011		rd		1110011		CSRRC	
csr				uimm		101		rd		1110011		CSRRWI	
csr				uimm		110		rd		1110011		CSRRSI	
csr				uimm		111		rd		1110011		CSRRCI	

RV32M Standard Extension

0000001			rs2		rs1		000		rd		0110011		MUL
0000001			rs2		rs1		001		rd		0110011		MULH
0000001			rs2		rs1		010		rd		0110011		MULHSU

0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

0000001	rs2	rs1	000	rd	0111011	MULW
0000001	rs2	rs1	100	rd	0111011	DIVW
0000001	rs2	rs1	101	rd	0111011	DIVUW
0000001	rs2	rs1	110	rd	0111011	REMW
0000001	rs2	rs1	111	rd	0111011	REMUW

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	AMOODR.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	AMOODR.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

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

RV32D Standard Extension

imm[11:0]				rs1	011	rd	0000111	FLD
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	FSD	
rs3	01	rs2	rs1	rm	rd	1000011	FMADD.D	
rs3	01	rs2	rs1	rm	rd	1000111	FMSUB.D	
rs3	01	rs2	rs1	rm	rd	1001011	FNMSUB.D	
rs3	01	rs2	rs1	rm	rd	1001111	FNMADD.D	
0000001		rs2	rs1	rm	rd	1010011	FADD.D	
0000101		rs2	rs1	rm	rd	1010011	FSUB.D	
0001001		rs2	rs1	rm	rd	1010011	FMUL.D	
0001101		rs2	rs1	rm	rd	1010011	FDIV.D	
0101101		00000	rs1	rm	rd	1010011	FSQRT.D	
0010001		rs2	rs1	000	rd	1010011	FSGNJ.D	
0010001		rs2	rs1	001	rd	1010011	FSGNJN.D	
0010001		rs2	rs1	010	rd	1010011	FSGNJX.D	
0010101		rs2	rs1	000	rd	1010011	FMIN.D	
0010101		rs2	rs1	001	rd	1010011	FMAX.D	
0100000		00001	rs1	rm	rd	1010011	FCVT.S.D	
0100001		00000	rs1	rm	rd	1010011	FCVT.D.S	
1010001		rs2	rs1	010	rd	1010011	FEQ.D	
1010001		rs2	rs1	001	rd	1010011	FLT.D	
1010001		rs2	rs1	000	rd	1010011	FLE.D	
1110001		00000	rs1	001	rd	1010011	FCLASS.D	
1100001		00000	rs1	rm	rd	1010011	FCVT.W.D	
1100001		00001	rs1	rm	rd	1010011	FCVT.WU.D	
1101001		00000	rs1	rm	rd	1010011	FCVT.D.W	
1101001		00001	rs1	rm	rd	1010011	FCVT.D.WU	

RV64D Standard Extension (in addition to RV32D)

1100001		00010	rs1	rm	rd	1010011	FCVT.L.D
---------	--	-------	-----	----	----	---------	----------

1100001				00011		rs1		rm		rd		1010011		FCVT.LU.D	
1110001				00000		rs1		000		rd		1010011		FMV.X.D	
1101001				00010		rs1		rm		rd		1010011		FCVT.D.L	
1101001				00011		rs1		rm		rd		1010011		FCVT.D.LU	
1111001				00000		rs1		000		rd		1010011		FMV.D.X	
31	27	26	25	24	20	19	15	14	12	11		7	6		0
funct7				rs2		rs1		funct3		rd		opcode		R-type	
rs3		funct2		rs2		rs1		funct3		rd		opcode		R4-type	
imm[11:0]						rs1		funct3		rd		opcode		I-type	
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type	

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

RV32Zfh Standard Extension						
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 40 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 40. 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 31. Extending RISC-V

In addition to supporting standard general-purpose software development, another goal of RISC-V is to provide a basis for more specialized instruction-set extensions or more customized accelerators. The instruction encoding spaces and optional variable-length instruction encoding are designed to make it easier to leverage software development effort for the standard ISA toolchain when building more customized processors. For example, the intent is to continue to provide full software support for implementations that only use the standard I base, perhaps together with many non-standard instruction-set extensions.

This chapter describes various ways in which the base RISC-V ISA can be extended, together with the scheme for managing instruction-set extensions developed by independent groups. This volume only deals with the unprivileged ISA, although the same approach and terminology is used for supervisor-level extensions described in the second volume.

31.1. Extension Terminology

This section defines some standard terminology for describing RISC-V extensions.

31.1.1. Standard versus Non-Standard Extension

Any RISC-V processor implementation must support a base integer ISA (RV32I, RV32E, RV64I, RV64E, or RV128I). In addition, an implementation may support one or more extensions. We divide extensions into two broad categories: *standard* versus *non-standard*.

- A standard extension is one that is generally useful and that is designed to not conflict with any other standard extension. Currently, "MAFDQLCBTPV", described in other chapters of this manual, are either complete or planned standard extensions.
- A non-standard extension may be highly specialized and may conflict with other standard or non-standard extensions. We anticipate a wide variety of non-standard extensions will be developed over time, with some eventually being promoted to standard extensions.

31.1.2. Instruction Encoding Spaces and Prefixes

An instruction encoding space is some number of instruction bits within which a base ISA or ISA extension is encoded. RISC-V supports varying instruction lengths, but even within a single instruction length, there are various sizes of encoding space available. For example, the base ISAs are defined within a 30-bit encoding space (bits 31-2 of the 32-bit instruction), while the atomic extension "A" fits within a 25-bit encoding space (bits 31-7).

We use the term *prefix* to refer to the bits to the *right* of an instruction encoding space (since instruction fetch in RISC-V is little-endian, the bits to the right are stored at earlier memory addresses, hence form a prefix in instruction-fetch order). The prefix for the standard base ISA encoding is the two-bit "11" field held in bits 1-0 of the 32-bit word, while the prefix for the standard atomic extension

"A" is the seven-bit "0101111" field held in bits 6-0 of the 32-bit word representing the AMO major opcode. A quirk of the encoding format is that the 3-bit funct3 field used to encode a minor opcode is not contiguous with the major opcode bits in the 32-bit instruction format, but is considered part of the prefix for 22-bit instruction spaces.

Although an instruction encoding space could be of any size, adopting a smaller set of common sizes simplifies packing independently developed extensions into a single global encoding. Table 41 gives the suggested sizes for RISC-V.

Table 41. Suggested standard RISC-V instruction encoding space sizes.

Size	Usage	# Available in standard instruction length			
		16-bit	32-bit	48-bit	64-bit
14-bit	Quadrant of compressed 16-bit encoding	3			
22-bit	Minor opcode in base 32-bit encoding		2^8	2^{20}	2^{35}
25-bit	Major opcode in base 32-bit encoding		32	2^{17}	2^{32}
30-bit	Quadrant of base 32-bit encoding		1	2^{12}	2^{27}
32-bit	Minor opcode in 48-bit encoding			2^{10}	2^{25}
37-bit	Major opcode in 48-bit encoding			32	2^{20}
40-bit	Quadrant of 48-bit encoding			4	2^{17}
45-bit	Sub-minor opcode in 64-bit encoding				2^{12}
48-bit	Minor opcode in 64-bit encoding				2^9
52-bit	Major opcode in 64-bit encoding				32

31.1.3. Greenfield versus Brownfield Extensions

We use the term *greenfield extension* to describe an extension that begins populating a new instruction encoding space, and hence can only cause encoding conflicts at the prefix level. We use the term *brownfield extension* to describe an extension that fits around existing encodings in a previously defined instruction space. A brownfield extension is necessarily tied to a particular greenfield parent encoding, and there may be multiple brownfield extensions to the same greenfield parent encoding. For example, the base ISAs are greenfield encodings of a 30-bit instruction space, while the FDQ floating-point extensions are all brownfield extensions adding to the parent base ISA 30-bit encoding space.

Note that we consider the standard A extension to have a greenfield encoding as it defines a new previously empty 25-bit encoding space in the leftmost bits of the full 32-bit base instruction encoding, even though its standard prefix locates it within the 30-bit encoding space of its parent base ISA. Changing only its single 7-bit prefix could move the A extension to a different 30-bit encoding space while only worrying about conflicts at the prefix level, not within the encoding space itself.

Table 42. Two-dimensional characterization of standard instruction-set extensions.

	Adds state	No new state
Greenfield	RV32I(30), RV64I(30)	A(25)
Brownfield	F(I), D(F), Q(D)	M(I)

Table 42 shows the bases and standard extensions placed in a simple two-dimensional taxonomy. One axis is whether the extension is greenfield or brownfield, while the other axis is whether the extension adds architectural state. For greenfield extensions, the size of the instruction encoding space is given in parentheses. For brownfield extensions, the name of the extension (greenfield or brownfield) it builds upon is given in parentheses. Additional user-level architectural state usually implies changes to the supervisor-level system or possibly to the standard calling convention.

Note that RV64I is not considered an extension of RV32I, but a different complete base encoding.

31.1.4. Standard-Compatible Global Encodings

A complete or *global* encoding of an ISA for an actual RISC-V implementation must allocate a unique non-conflicting prefix for every included instruction encoding space. The bases and every standard extension have each had a standard prefix allocated to ensure they can all coexist in a global encoding.

A *standard-compatible* global encoding is one where the base and every included standard extension have their standard prefixes. A standard-compatible global encoding can include non-standard extensions that do not conflict with the included standard extensions. A standard-compatible global encoding can also use standard prefixes for non-standard extensions if the associated standard extensions are not included in the global encoding. In other words, a standard extension must use its standard prefix if included in a standard-compatible global encoding, but otherwise its prefix is free to be reallocated. These constraints allow a common toolchain to target the standard subset of any RISC-V standard-compatible global encoding.

31.1.5. Guaranteed Non-Standard Encoding Space

To support development of proprietary custom extensions, portions of the encoding space are guaranteed to never be used by standard extensions.

31.2. RISC-V Extension Design Philosophy

We intend to support a large number of independently developed extensions by encouraging extension developers to operate within instruction encoding spaces, and by providing tools to pack these into a standard-compatible global encoding by allocating unique prefixes. Some extensions are more naturally implemented as brownfield augmentations of existing extensions, and will share whatever prefix is allocated to their parent greenfield extension. The standard extension prefixes avoid spurious incompatibilities in the encoding of core functionality, while allowing custom packing of more esoteric extensions.

This capability of repacking RISC-V extensions into different standard-compatible global encodings can be used in a number of ways.

One use-case is developing highly specialized custom accelerators, designed to run kernels from important application domains. These might want to drop all but the base integer ISA and add in only the extensions that are required for the task in hand. The base ISAs have been designed to place minimal requirements on a hardware implementation, and has been encoded to use only a small fraction of a 32-bit instruction encoding space.

Another use-case is to build a research prototype for a new type of instruction-set extension. The researchers might not want to expend the effort to implement a variable-length instruction-fetch unit, and so would like to prototype their extension using a simple 32-bit fixed-width instruction encoding. However, this new extension might be too large to coexist with standard extensions in the 32-bit space. If the research experiments do not need all of the standard extensions, a standard-compatible global encoding might drop the unused standard extensions and reuse their prefixes to place the proposed extension in a non-standard location to simplify engineering of the research prototype. Standard tools will still be able to target the base and any standard extensions that are present to reduce development time. Once the instruction-set extension has been evaluated and refined, it could then be made available for packing into a larger variable-length encoding space to avoid conflicts with all standard extensions.

The following sections describe increasingly sophisticated strategies for developing implementations with new instruction-set extensions. These are mostly intended for use in highly customized, educational, or experimental architectures rather than for the main line of RISC-V ISA development.

31.3. Extensions within fixed-width 32-bit instruction format

In this section, we discuss adding extensions to implementations that only support the base fixed-width 32-bit instruction format.



We anticipate the simplest fixed-width 32-bit encoding will be popular for many restricted accelerators and research prototypes.

31.3.1. Available 30-bit instruction encoding spaces

In the standard encoding, three of the available 30-bit instruction encoding spaces (those with 2-bit prefixes **00**, **01**, and **10**) are used to enable the optional compressed instruction extension. However, if the compressed instruction-set extension is not required, then these three further 30-bit encoding spaces become available. This quadruples the available encoding space within the 32-bit format.

31.3.2. Available 25-bit instruction encoding spaces

A 25-bit instruction encoding space corresponds to a major opcode in the base and standard extension encodings.

There are four major opcodes expressly designated for custom extensions [Table 39](#), each of which represents a 25-bit encoding space. Two of these are reserved for eventual use in the RV128 base encoding (will be OP-IMM-64 and OP-64), but can be used for non-standard extensions for RV32 and RV64.

The two major opcodes reserved for RV64 (OP-IMM-32 and OP-32) can also be used for non-standard extensions to RV32 only.

If an implementation does not require floating-point, then the seven major opcodes reserved for standard floating-point extensions (LOAD-FP, STORE-FP, MADD, MSUB, NMSUB, NMADD, OP-FP) can be reused for non-standard extensions. Similarly, the AMO major opcode can be reused if the standard atomic extensions are not required.

If an implementation does not require instructions longer than 32-bits, then an additional four major opcodes are available (those marked in gray in [Table 39](#)).

The base RV32I encoding uses only 11 major opcodes plus 3 reserved opcodes, leaving up to 18 available for extensions. The base RV64I encoding uses only 13 major opcodes plus 3 reserved opcodes, leaving up to 16 available for extensions.

31.3.3. Available 22-bit instruction encoding spaces

A 22-bit encoding space corresponds to a funct3 minor opcode space in the base and standard extension encodings. Several major opcodes have a funct3 field minor opcode that is not completely occupied, leaving available several 22-bit encoding spaces.

Usually a major opcode selects the format used to encode operands in the remaining bits of the instruction, and ideally, an extension should follow the operand format of the major opcode to simplify hardware decoding.

31.3.4. Other spaces

Smaller spaces are available under certain major opcodes, and not all minor opcodes are entirely filled.

31.4. Adding aligned 64-bit instruction extensions

The simplest approach to provide space for extensions that are too large for the base 32-bit fixed-width instruction format is to add naturally aligned 64-bit instructions. The implementation must still support the 32-bit base instruction format, but can require that 64-bit instructions are aligned on 64-bit boundaries to simplify instruction fetch, with a 32-bit NOP instruction used as alignment padding where necessary.

To simplify use of standard tools, the 64-bit instructions should be encoded as described in [Table 1](#). However, an implementation might choose a non-standard instruction-length encoding for 64-bit

instructions, while retaining the standard encoding for 32-bit instructions. For example, if compressed instructions are not required, then a 64-bit instruction could be encoded using one or more zero bits in the first two bits of an instruction.



We anticipate processor generators that produce instruction-fetch units capable of automatically handling any combination of supported variable-length instruction encodings.

31.5. Supporting VLIW encodings

Although RISC-V was not designed as a base for a pure VLIW machine, VLIW encodings can be added as extensions using several alternative approaches. In all cases, the base 32-bit encoding has to be supported to allow use of any standard software tools.

31.5.1. Fixed-size instruction group

The simplest approach is to define a single large naturally aligned instruction format (e.g., 128 bits) within which VLIW operations are encoded. In a conventional VLIW, this approach would tend to waste instruction memory to hold NOPs, but a RISC-V-compatible implementation would have to also support the base 32-bit instructions, confining the VLIW code size expansion to VLIW-accelerated functions.

31.5.2. Encoded-Length Groups

Another approach is to use the standard length encoding from [Table 1](#) to encode parallel instruction groups, allowing NOPs to be compressed out of the VLIW instruction. For example, a 64-bit instruction could hold two 28-bit operations, while a 96-bit instruction could hold three 28-bit operations, and so on. Alternatively, a 48-bit instruction could hold one 42-bit operation, while a 96-bit instruction could hold two 42-bit operations, and so on.

This approach has the advantage of retaining the base ISA encoding for instructions holding a single operation, but has the disadvantage of requiring a new 28-bit or 42-bit encoding for operations within the VLIW instructions, and misaligned instruction fetch for larger groups. One simplification is to not allow VLIW instructions to straddle certain microarchitecturally significant boundaries (e.g., cache lines or virtual memory pages).

31.5.3. Fixed-Size Instruction Bundles

Another approach, similar to Itanium, is to use a larger naturally aligned fixed instruction bundle size (e.g., 128 bits) across which parallel operation groups are encoded. This simplifies instruction fetch, but shifts the complexity to the group execution engine. To remain RISC-V compatible, the base 32-bit instruction would still have to be supported.

31.5.4. End-of-Group bits in Prefix

None of the above approaches retains the RISC-V encoding for the individual operations within a VLIW instruction. Yet another approach is to repurpose the two prefix bits in the fixed-width 32-bit encoding. One prefix bit can be used to signal "end-of-group" if set, while the second bit could indicate execution under a predicate if clear. Standard RISC-V 32-bit instructions generated by tools unaware of the VLIW extension would have both prefix bits set (11) and thus have the correct semantics, with each instruction at the end of a group and not predicated.

The main disadvantage of this approach is that the base ISAs lack the complex predication support usually required in an aggressive VLIW system, and it is difficult to add space to specify more predicate registers in the standard 30-bit encoding space.

DRAFT

Chapter 32. 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.

32.1. Case Sensitivity

The ISA naming strings are case insensitive.

32.2. Base Integer ISA

RISC-V ISA strings begin with either RV32I, RV32E, RV64I, RV64E, or RV128I indicating the supported address space size in bits for the base integer ISA.

32.3. 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.

32.4. 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. 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 "RV64I1pOM1pOA1pOF1pOD1pO", 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".

32.5. Underscores

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

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 a 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.

32.6. Additional Standard Extension Names

Standard extensions can also be named using a single "Z" followed by an alphabetical name and an optional version number. For example, "Zifencei" names the instruction-fetch fence extension described in [Chapter 3](#); "Zifencei2" and "Zifencei2p0" name version 2.0 of same.

The first letter following the "Z" conventionally indicates the most closely related alphabetical extension category, IMAFDQCVH. 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_Zam".

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".

32.7. Supervisor-level Instruction-Set Extensions

Standard extensions that extend the supervisor-level virtual-memory architecture are prefixed with the letters "Sv", followed by an alphabetical name and an optional version number, or by a numeric name with no version number. Other standard extensions that extend the supervisor-level architecture are prefixed with the letters "Ss", followed by an alphabetical name and an optional version number. Such extensions are defined in Volume II.

Standard supervisor-level extensions should be listed after standard unprivileged extensions. If multiple supervisor-level extensions are listed, they should be ordered alphabetically.

32.8. Hypervisor-level Instruction-Set Extensions

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.

32.9. Machine-level Instruction-Set Extensions

Standard machine-level instruction-set extensions are prefixed with the letters "Sm".

Standard machine-level extensions should be listed after standard lesser-privileged extensions. If multiple machine-level extensions are listed, they should be ordered alphabetically.

32.10. Non-Standard Extension Names

Non-standard extensions are named using a single "X" followed by an alphabetical name and an optional version number. For example, "Xhwacha" names the Hwacha vector-fetch ISA extension; "Xhwacha2" and "Xhwacha2p0" name version 2.0 of same.

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.

32.11. Subset Naming Convention

Table 43 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 43. Standard ISA extension names.

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	Zmmul

Subset	Name	Implies
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	
Packed-SIMD Extensions	P	
Vector Extension	V	D
Hypervisor Extension	H	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension "def"	Ssdef	
Standard Machine-Level Extensions		
Machine-level extension "jkl"	Smjkl	
Non-Standard Extensions		
Non-standard extension "mno"	Xmno	

Chapter 33. History and Acknowledgments

33.1. "Why Develop a new ISA?" Rationale from Berkeley Group

We developed RISC-V to support our own needs in research and education, where our group is particularly interested in actual hardware implementations of research ideas (we have completed eleven different silicon fabrications of RISC-V since the first edition of this specification), and in providing real implementations for students to explore in classes (RISC-V processor RTL designs have been used in multiple undergraduate and graduate classes at Berkeley). In our current research, we are especially interested in the move towards specialized and heterogeneous accelerators, driven by the power constraints imposed by the end of conventional transistor scaling. We wanted a highly flexible and extensible base ISA around which to build our research effort.

A question we have been repeatedly asked is "Why develop a new ISA?" The biggest obvious benefit of using an existing commercial ISA is the large and widely supported software ecosystem, both development tools and ported applications, which can be leveraged in research and teaching. Other benefits include the existence of large amounts of documentation and tutorial examples. However, our experience of using commercial instruction sets for research and teaching is that these benefits are smaller in practice, and do not outweigh the disadvantages:

- **Commercial ISAs are proprietary.** Except for SPARC V8, which is an open IEEE standard ([*IEEE Standard for a 32-Bit Microprocessor*, 1994](#)), most owners of commercial ISAs carefully guard their intellectual property and do not welcome freely available competitive implementations. This is much less of an issue for academic research and teaching using only software simulators, but has been a major concern for groups wishing to share actual RTL implementations. It is also a major concern for entities who do not want to trust the few sources of commercial ISA implementations, but who are prohibited from creating their own clean room implementations. We cannot guarantee that all RISC-V implementations will be free of third-party patent infringements, but we can guarantee we will not attempt to sue a RISC-V implementor.
- **Commercial ISAs are only popular in certain market domains.** The most obvious examples at time of writing are that the ARM architecture is not well supported in the server space, and the Intel x86 architecture (or for that matter, almost every other architecture) is not well supported in the mobile space, though both Intel and ARM are attempting to enter each other's market segments. Another example is ARC and Tensilica, which provide extensible cores but are focused on the embedded space. This market segmentation dilutes the benefit of supporting a particular commercial ISA as in practice the software ecosystem only exists for certain domains, and has to be built for others.
- **Commercial ISAs come and go.** Previous research infrastructures have been built around commercial ISAs that are no longer popular (SPARC, MIPS) or even no longer in production (Alpha). These lose the benefit of an active software ecosystem, and the lingering intellectual property issues around the ISA and supporting tools interfere with the ability of interested third parties to continue supporting the ISA. An open ISA might also lose popularity, but any interested party can continue using and developing the ecosystem.
- **Popular commercial ISAs are complex.** The dominant commercial ISAs (x86 and ARM) are both very complex to implement in hardware to the level of supporting common software stacks and operating systems. Worse, nearly all the complexity is due to bad, or at least outdated, ISA design decisions rather than features that truly improve efficiency.

- **Commercial ISAs alone are not enough to bring up applications.** Even if we expend the effort to implement a commercial ISA, this is not enough to run existing applications for that ISA. Most applications need a complete ABI (application binary interface) to run, not just the user-level ISA. Most ABIs rely on libraries, which in turn rely on operating system support. To run an existing operating system requires implementing the supervisor-level ISA and device interfaces expected by the OS. These are usually much less well-specified and considerably more complex to implement than the user-level ISA.
- **Popular commercial ISAs were not designed for extensibility.** The dominant commercial ISAs were not particularly designed for extensibility, and as a consequence have added considerable instruction encoding complexity as their instruction sets have grown. Companies such as Tensilica (acquired by Cadence) and ARC (acquired by Synopsys) have built ISAs and toolchains around extensibility, but have focused on embedded applications rather than general-purpose computing systems.
- **A modified commercial ISA is a new ISA.** One of our main goals is to support architecture research, including major ISA extensions. Even small extensions diminish the benefit of using a standard ISA, as compilers have to be modified and applications rebuilt from source code to use the extension. Larger extensions that introduce new architectural state also require modifications to the operating system. Ultimately, the modified commercial ISA becomes a new ISA, but carries along all the legacy baggage of the base ISA.

Our position is that the ISA is perhaps the most important interface in a computing system, and there is no reason that such an important interface should be proprietary. The dominant commercial ISAs are based on instruction-set concepts that were already well known over 30 years ago. Software developers should be able to target an open standard hardware target, and commercial processor designers should compete on implementation quality.

We are far from the first to contemplate an open ISA design suitable for hardware implementation. We also considered other existing open ISA designs, of which the closest to our goals was the OpenRISC architecture ([OpenCores, 2012](#)). We decided against adopting the OpenRISC ISA for several technical reasons:

- OpenRISC has condition codes and branch delay slots, which complicate higher performance implementations.
- OpenRISC uses a fixed 32-bit encoding and 16-bit immediates, which precludes a denser instruction encoding and limits space for later expansion of the ISA.
- OpenRISC does not support the 2008 revision to the IEEE 754 floating-point standard.
- The OpenRISC 64-bit design had not been completed when we began.

By starting from a clean slate, we could design an ISA that met all of our goals, though of course, this took far more effort than we had planned at the outset. We have now invested considerable effort in building up the RISC-V ISA infrastructure, including documentation, compiler tool chains, operating system ports, reference ISA simulators, FPGA implementations, efficient ASIC implementations, architecture test suites, and teaching materials. Since the last edition of this manual, there has been considerable uptake of the RISC-V ISA in both academia and industry, and we have created the non-profit RISC-V Foundation to protect and promote the standard. The RISC-V Foundation website at riscv.org contains the latest information on the Foundation membership and various open-source projects using RISC-V.

33.2. History from Revision 1.0 of ISA manual

The RISC-V ISA and instruction-set manual builds upon several earlier projects. Several aspects of the supervisor-level machine and the overall format of the manual date back to the TO (Torrent-O) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. TO was a vector processor based on the MIPS-II ISA, with Krste Asanović as main architect and RTL designer, and Brian Kingsbury and Bertrand Irrisou as principal VLSI implementors. David Johnson at ICSI was a major contributor to the TO ISA design, particularly supervisor mode, and to the manual text. John Hauser also provided considerable feedback on the TO ISA design.

The Scale (Software-Controlled Architecture for Low Energy) project at MIT, begun in 2000, built upon the TO project infrastructure, refined the supervisor-level interface, and moved away from the MIPS scalar ISA by dropping the branch delay slot. Ronny Krashinsky and Christopher Batten were the principal architects of the Scale Vector-Thread processor at MIT, while Mark Hampton ported the GCC-based compiler infrastructure and tools for Scale.

A lightly edited version of the TO MIPS scalar processor specification (MIPS-6371) was used in teaching a new version of the MIT 6.371 Introduction to VLSI Systems class in the Fall 2002 semester, with Chris Terman and Krste Asanović as lecturers. Chris Terman contributed most of the lab material for the class (there was no TA!). The 6.371 class evolved into the trial 6.884 Complex Digital Design class at MIT, taught by Arvind and Krste Asanović in Spring 2005, which became a regular Spring class 6.375. A reduced version of the Scale MIPS-based scalar ISA, named SMIPS, was used in 6.884/6.375. Christopher Batten was the TA for the early offerings of these classes and developed a considerable amount of documentation and lab material based around the SMIPS ISA. This same SMIPS lab material was adapted and enhanced by TA Yunsup Lee for the UC Berkeley Fall 2009 CS250 VLSI Systems Design class taught by John Wawrzynek, Krste Asanović, and John Lazzaro.

The Maven (Malleable Array of Vector-thread ENgines) project was a second-generation vector-thread architecture. Its design was led by Christopher Batten when he was an Exchange Scholar at UC Berkeley starting in summer 2007. Hidetaka Aoki, a visiting industrial fellow from Hitachi, gave considerable feedback on the early Maven ISA and microarchitecture design. The Maven infrastructure was based on the Scale infrastructure but the Maven ISA moved further away from the MIPS ISA variant defined in Scale, with a unified floating-point and integer register file. Maven was designed to support experimentation with alternative data-parallel accelerators. Yunsup Lee was the main implementor of the various Maven vector units, while Rimas Avižienis was the main implementor of the various Maven scalar units. Yunsup Lee and Christopher Batten ported GCC to work with the new Maven ISA. Christopher Celio provided the initial definition of a traditional vector instruction set ("Flood") variant of Maven.

Based on experience with all these previous projects, the RISC-V ISA definition was begun in Summer 2010, with Andrew Waterman, Yunsup Lee, Krste Asanović, and David Patterson as principal designers. An initial version of the RISC-V 32-bit instruction subset was used in the UC Berkeley Fall 2010 CS250 VLSI Systems Design class, with Yunsup Lee as TA. RISC-V is a clean break from the earlier MIPS-inspired designs. John Hauser contributed to the floating-point ISA definition, including the sign-injection instructions and a register encoding scheme that permits internal recoding of floating-point values.

33.3. History from Revision 2.0 of ISA manual

Multiple implementations of RISC-V processors have been completed, including several silicon fabrications, as shown in [Fabricated RISC-V testchips table](#).

Name	Tapeout Date	Process	ISA
Raven-1	May 29, 2011	ST 28nm FDSOI	RV64G1_Xhwacha1
EOS14	April 1, 2012	IBM 45nm SOI	RV64G1p1_Xhwacha2
EOS16	August 17, 2012	IBM 45nm SOI	RV64G1p1_Xhwacha2
Raven-2	August 22, 2012	ST 28nm FDSOI	RV64G1p1_Xhwacha2
EOS18	February 6, 2013	IBM 45nm SOI	RV64G1p1_Xhwacha2
EOS20	July 3, 2013	IBM 45nm SOI	RV64G1p99_Xhwacha2
Raven-3	September 26, 2013	ST 28nm SOI	RV64G1p99_Xhwacha2
EOS22	March 7, 2014	IBM 45nm SOI	RV64G1p9999_Xhwacha3

The first RISC-V processors to be fabricated were written in Verilog and manufactured in a pre-production FDSOI technology from ST as the Raven-1 testchip in 2011. Two cores were developed by Yunsup Lee and Andrew Waterman, advised by Krste Asanović, and fabricated together: 1) an RV64 scalar core with error-detecting flip-flops, and 2) an RV64 core with an attached 64-bit floating-point vector unit. The first microarchitecture was informally known as "TrainWreck", due to the short time available to complete the design with immature design libraries.

Subsequently, a clean microarchitecture for an in-order decoupled RV64 core was developed by Andrew Waterman, Rimas Avižienis, and Yunsup Lee, advised by Krste Asanović, and, continuing the railway theme, was codenamed "Rocket" after George Stephenson's successful steam locomotive design. Rocket was written in Chisel, a new hardware design language developed at UC Berkeley. The IEEE floating-point units used in Rocket were developed by John Hauser, Andrew Waterman, and Brian Richards. Rocket has since been refined and developed further, and has been fabricated two more times in FDSOI (Raven-2, Raven-3), and five times in IBM SOI technology (EOS14, EOS16, EOS18, EOS20, EOS22) for a photonics project. Work is ongoing to make the Rocket design available as a parameterized RISC-V processor generator.

EOS14-EOS22 chips include early versions of Hwacha, a 64-bit IEEE floating-point vector unit, developed by Yunsup Lee, Andrew Waterman, Huy Vo, Albert Ou, Quan Nguyen, and Stephen Twigg, advised by Krste Asanović. EOS16-EOS22 chips include dual cores with a cache-coherence protocol developed by Henry Cook and Andrew Waterman, advised by Krste Asanović. EOS14 silicon has successfully run at 1.25 GHz. EOS16 silicon suffered from a bug in the IBM pad libraries. EOS18 and EOS20 have successfully run at 1.35 GHz.

Contributors to the Raven testchips include Yunsup Lee, Andrew Waterman, Rimas Avižienis, Brian Zimmer, Jaehwa Kwak, Ruzica Jevtić, Milovan Blagojević, Alberto Puggelli, Steven Bailey, Ben Keller, Pi-Feng Chiu, Brian Richards, Borivoje Nikolić, and Krste Asanović.

Contributors to the EOS testchips include Yunsup Lee, Rimas Avižienis, Andrew Waterman, Henry Cook, Huy Vo, Daiwei Li, Chen Sun, Albert Ou, Quan Nguyen, Stephen Twigg, Vladimir Stojanović, and Krste Asanović.

Andrew Waterman and Yunsup Lee developed the C++ ISA simulator "Spike", used as a golden model in development and named after the golden spike used to celebrate completion of the US

transcontinental railway. Spike has been made available as a BSD open-source project.

Andrew Waterman completed a Master's thesis with a preliminary design of the RISC-V compressed instruction set ([Waterman, 2011](#)).

Various FPGA implementations of the RISC-V have been completed, primarily as part of integrated demos for the Par Lab project research retreats. The largest FPGA design has 3 cache-coherent RV64IMA processors running a research operating system. Contributors to the FPGA implementations include Andrew Waterman, Yunsup Lee, Rimas Avižienis, and Krste Asanović.

RISC-V processors have been used in several classes at UC Berkeley. Rocket was used in the Fall 2011 offering of CS250 as a basis for class projects, with Brian Zimmer as TA. For the undergraduate CS152 class in Spring 2012, Christopher Celio used Chisel to write a suite of educational RV32 processors, named "Sodor" after the island on which "Thomas the Tank Engine" and friends live. The suite includes a microcoded core, an unpipelined core, and 2, 3, and 5-stage pipelined cores, and is publicly available under a BSD license. The suite was subsequently updated and used again in CS152 in Spring 2013, with Yunsup Lee as TA, and in Spring 2014, with Eric Love as TA. Christopher Celio also developed an out-of-order RV64 design known as BOOM (Berkeley Out-of-Order Machine), with accompanying pipeline visualizations, that was used in the CS152 classes. The CS152 classes also used cache-coherent versions of the Rocket core developed by Andrew Waterman and Henry Cook.

Over the summer of 2013, the RoCC (Rocket Custom Coprocessor) interface was defined to simplify adding custom accelerators to the Rocket core. Rocket and the RoCC interface were used extensively in the Fall 2013 CS250 VLSI class taught by Jonathan Bachrach, with several student accelerator projects built to the RoCC interface. The Hwacha vector unit has been rewritten as a RoCC coprocessor.

Two Berkeley undergraduates, Quan Nguyen and Albert Ou, have successfully ported Linux to run on RISC-V in Spring 2013.

Colin Schmidt successfully completed an LLVM backend for RISC-V 2.0 in January 2014.

Darius Rad at Bluespec contributed soft-float ABI support to the GCC port in March 2014.

John Hauser contributed the definition of the floating-point classification instructions.

We are aware of several other RISC-V core implementations, including one in Verilog by Tommy Thorn, and one in Bluespec by Rishiyur Nikhil.

33.4. Acknowledgments

Thanks to Christopher F. Batten, Preston Briggs, Christopher Celio, David Chisnall, Stefan Freudenberger, John Hauser, Ben Keller, Rishiyur Nikhil, Michael Taylor, Tommy Thorn, and Robert Watson for comments on the draft ISA version 2.0 specification.

33.5. History from Revision 2.1

Uptake of the RISC-V ISA has been very rapid since the introduction of the frozen version 2.0 in May 2014, with too much activity to record in a short history section such as this. Perhaps the most

important single event was the formation of the non-profit RISC-V Foundation in August 2015. The Foundation will now take over stewardship of the official RISC-V ISA standard, and the official website riscv.org is the best place to obtain news and updates on the RISC-V standard.

33.6. Acknowledgments

Thanks to Scott Beamer, Allen J. Baum, Christopher Celio, David Chisnall, Paul Clayton, Palmer Dabbelt, Jan Gray, Michael Hamburg, and John Hauser for comments on the version 2.0 specification.

33.7. History from Revision 2.2

33.8. Acknowledgments

Thanks to Jacob Bachmeyer, Alex Bradbury, David Horner, Stefan O'Rear, and Joseph Myers for comments on the version 2.1 specification.

33.9. History for Revision 2.3

Uptake of RISC-V continues at a breakneck pace.

John Hauser and Andrew Waterman contributed a hypervisor ISA extension based upon a proposal from Paolo Bonzini.

Daniel Lustig, Arvind, Krste Asanović, Shaked Flur, Paul Loewenstein, Yatin Manerkar, Luc Maranget, Margaret Martonosi, Vijayanand Nagarajan, Rishiyur Nikhil, Jonas Oberhauser, Christopher Pulte, Jose Renau, Peter Sewell, Susmit Sarkar, Caroline Trippel, Muralidaran Vijayaraghavan, Andrew Waterman, Derek Williams, Andrew Wright, and Sizhuo Zhang contributed the memory consistency model.

33.10. Funding

Development of the RISC-V architecture and implementations has been partially funded by the following sponsors.

- **Par Lab:** Research supported by Microsoft (Award # 024263) and Intel (Award # 024894) funding and by matching funding by U.C. Discovery (Award # DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.
- **Project Isis:** DoE Award DE-SC0003624.

- **ASPIRE Lab:** DARPA PERFECT program, Award HRO011-12-2-0016. DARPA POEM program Award HRO011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Hewlett Packard Enterprise, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

The content of this paper does not necessarily reflect the position or the policy of the US government and no official endorsement should be inferred.

DRAFT

Appendix A: RVWMO Explanatory Material, Version 0.1

This section provides more explanation for RVWMO [Chapter 17](#), 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 17](#) and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in [Section A.7](#). Any other discrepancies are unintentional.

A.1. Why RVWMO?

Memory consistency models fall along a loose spectrum from weak to strong. Weak memory models allow more hardware implementation flexibility and deliver arguably better performance, performance per watt, power, scalability, and hardware verification overheads than strong models, at the expense of a more complex programming model. Strong models provide simpler programming models, but at the cost of imposing more restrictions on the kinds of (non-speculative) hardware optimizations that can be performed in the pipeline and in the memory system, and in turn imposing some cost in terms of power, area overhead, and verification burden.

RISC-V has chosen the RVWMO memory model, a variant of release consistency. This places it in between the two extremes of the memory model spectrum. The RVWMO memory model enables architects to build simple implementations, aggressive implementations, implementations embedded deeply inside a much larger system and subject to complex memory system interactions, or any number of other possibilities, all while simultaneously being strong enough to support programming language memory models at high performance.

To facilitate the porting of code from other architectures, some hardware implementations may choose to implement the Ztso extension, which provides stricter RVTSO ordering semantics by default. Code written for RVWMO is automatically and inherently compatible with RVTSO, but code written assuming RVTSO is not guaranteed to run correctly on RVWMO implementations. In fact, most RVWMO implementations will (and should) simply refuse to run RVTSO-only binaries. Each implementation must therefore choose whether to prioritize compatibility with RVTSO code (e.g., to facilitate porting from x86) or whether to instead prioritize compatibility with other RISC-V cores implementing RVWMO.

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

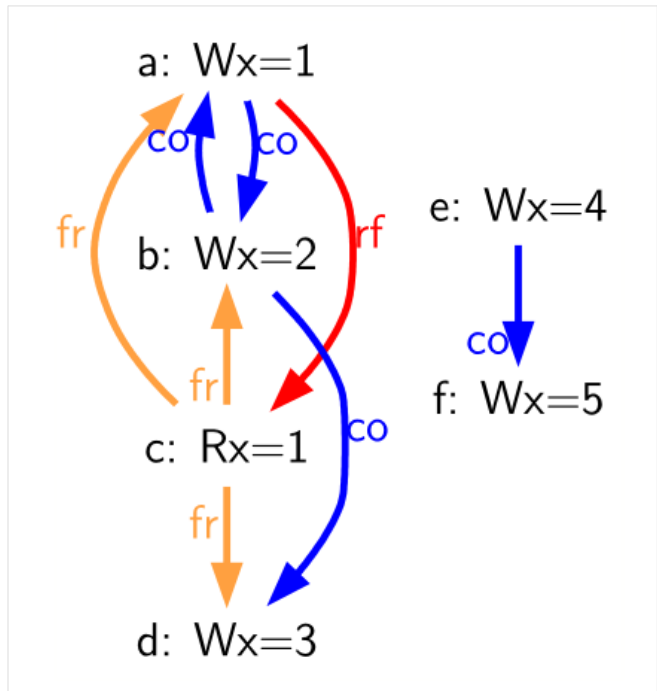
A.2. Litmus Tests

The explanations in this chapter make use of *litmus tests*, or small programs designed to test or highlight one particular aspect of a memory model. [Litmus sample](#) shows an example of a litmus test with two harts. As a convention for this figure and for all figures that follow in this chapter, we assume that `s0-s2` are pre-set to the same value in all harts and that `s0` holds the address labeled `x`, `s1` holds `y`, and `s2` holds `z`, where `x`, `y`, and `z` are disjoint memory locations aligned to 8 byte boundaries. All other

registers and all referenced memory locations are presumed to be initialized to zero. Each figure shows the litmus test code on the left, and a visualization of one particular valid or invalid execution on the right.

Table 44. 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 45. A key for the litmus test diagrams drawn in this appendix

Edge	Full Name (and explanation)
rf	Reads From (from each store to the loads that return a value written by that store)
co	Coherence (a total order on the stores to each address)
fr	From-Reads (from each load to co-successors of the store from which the load returned a value)
ppo	Preserved Program Order
fence	Orderings enforced by a FENCE instruction
addr	Address Dependency
ctrl	Control Dependency
data	Data Dependency

The diagram shown to the right of each litmus test shows a visual representation of the particular execution candidate being considered. These diagrams use a notation that is common in the memory model literature for constraining the set of possible global memory orders that could produce the execution in question. It is also the basis for the *herd* models presented in Section B.2. This notation is explained in Table 45. Of the listed relations, rf edges between harts, co edges, fr edges, and ppo edges directly constrain the global memory order (as do fence, addr, data, and some ctrl edges, via ppo). Other edges (such as intra-hart rf edges) are informative but do not constrain the global memory order.

For example, in Litmus sample, **a0=1** could occur only if one of the following were true:

- (b) appears before (a) in global memory order (and in the coherence order co). However, this violates RVWMO PPO rule **ppo:→st**. The co edge from (b) to (a) highlights this contradiction.
- (a) appears before (b) in global memory order (and in the coherence order co). However, in this case, the Load Value Axiom would be violated, because (a) is not the latest matching store prior to (c) in program order. The fr edge from (c) to (b) highlights this contradiction.

Since neither of these scenarios satisfies the RVWMO axioms, the outcome **a0=1** is forbidden.

Beyond what is described in this appendix, a suite of more than seven thousand litmus tests is available at github.com/litmus-tests/litmus-tests-riscv.



The litmus tests repository also provides instructions on how to run the litmus tests on RISC-V hardware and how to compare the results with the operational and axiomatic models.

In the future, we expect to adapt these memory model litmus tests for use as part of the RISC-V compliance test suite as well.

A.3. Explaining the RVWMO Rules

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

A.3.1. Preserved Program Order and Global Memory Order

Preserved program order represents the subset of program order that must be respected within the global memory order. Conceptually, events from the same hart that are ordered by preserved program order must appear in that order from the perspective of other harts and/or observers. Events from the same hart that are not ordered by preserved program order, on the other hand, may appear reordered from the perspective of other harts and/or observers.

Informally, the global memory order represents the order in which loads and stores perform. The formal memory model literature has moved away from specifications built around the concept of performing, but the idea is still useful for building up informal intuition. A load is said to have performed when its return value is determined. A store is said to have performed not when it has executed inside the pipeline, but rather only when its value has been propagated to globally visible memory. In this sense, the global memory order also represents the contribution of the coherence protocol and/or the rest of the memory system to interleave the (possibly reordered) memory accesses being issued by each hart into a single total order agreed upon by all harts.

The order in which loads perform does not always directly correspond to the relative age of the values those two loads return. In particular, a load b may perform before another load a to the same address (i.e., b may execute before a , and b may appear before a in the global memory order), but a may nevertheless return an older value than b . This discrepancy captures (among other things) the reordering effects of buffering placed between the core and memory. For example, b may have returned a value from a store in the store buffer, while a may have ignored that younger store and read an older value from memory instead. To account for this, at the time each load performs, the value it returns is determined by the load value axiom, not just strictly by determining the most recent store to the same address in the global memory order, as described below.

A.3.2. Load value axiom



Section 17.1.4.1: 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:

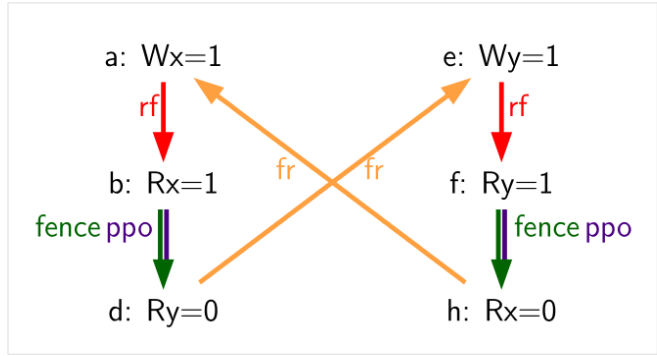
1. Stores that write that byte and that precede i in the global memory order
2. Stores that write that byte and that precede i in program order

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 46](#). 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 46. 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.

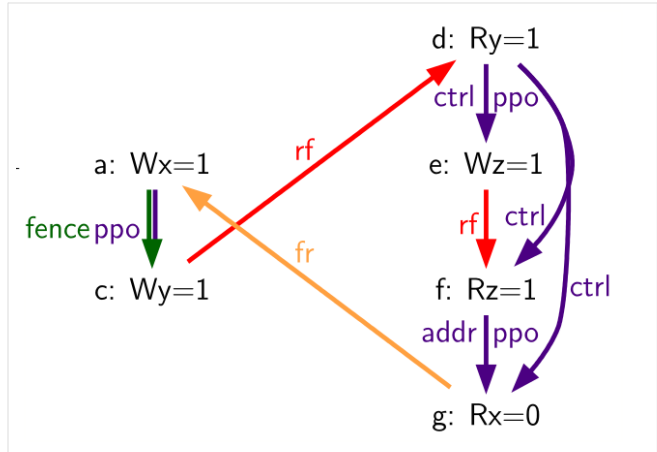
The global memory order must be a total order and cannot be cyclic, because a cycle would imply that every event in the cycle happens before itself, which is impossible. Therefore, the execution proposed above would be forbidden, and hence the addition of rule X would forbid implementations with store 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 47. 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 47. In this example, (d) is ordered before (e) because of the control dependency, and (f) is ordered before (g) because of the address dependency. However, (e) is *not* necessarily ordered before (f), even though (f) returns the value written by (e). This could correspond to the following sequence of events:

- (e) executes speculatively and enters the second hart's private store buffer (but does not drain to memory)
- (f) executes speculatively and forwards its return value 1 from (e) in the store buffer
- (g) executes speculatively and reads the value 0 from memory
- (a) executes, enters the first hart's private store buffer, and drains to memory
- (b) executes and retires
- (c) executes, enters the first hart's private store buffer, and drains to memory
- (d) executes and reads the value 1 from memory
- (e), (f), and (g) commit, since the speculation turned out to be correct
- (e) drains from the store buffer to memory

A.3.3. Atomicity axiom



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

The RISC-V architecture decouples the notion of atomicity from the notion of ordering. Unlike architectures such as TSO, RISC-V atomics under RVWMO do not impose any ordering requirements by default. Ordering semantics are only guaranteed by the PPO rules that otherwise apply.

RISC-V contains two types of atomics: AMOs and LR/SC pairs. These conceptually behave differently, in the following way. LR/SC behave as if the old value is brought up to the core, modified, and written back to memory, all while a reservation is held on that memory location. AMOs on the other hand conceptually behave as if they are performed directly in memory. AMOs are therefore inherently atomic, while LR/SC pairs are atomic in the slightly different sense that the memory location in question will not be modified by another hart during the time the original hart holds the reservation.

(a) lr.d aO, O(sO)	(a) lr.d aO, O(sO)	(a) lr.w aO, O(sO)	(a) lr.w aO, O(sO)
(b) sd t1, O(sO)	(b) sw t1, 4(sO)	(b) sw t1, 4(sO)	(b) sw t1, 4(sO)
(c) sc.d t3, t2, O(sO)	(c) sc.d t3, t2, O(sO)	(c) sc.w t3, t2, O(sO)	(c) addi sO, sO, 8
(d) sc.w t3, t2, 8(sO)			

Figure 4: In all four (independent) instances, the final store-conditional instruction is permitted but not guaranteed to succeed.

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 [litmus_lrdsdsc] 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 [litmus_lrdsdsc] as identical) will not be forced to fail a store-conditional instruction that happens to (falsely) share another portion of the same cache line as the memory location being held by the reservation.

The atomicity axiom also technically supports cases in which the LR and SC touch different addresses and/or use different access sizes; however, use cases for such behaviors are expected to be rare in practice. Likewise, scenarios in which stores from the same hart between an LR/SC pair actually overlap the memory location(s) referenced by the LR or SC are expected to be rare compared to scenarios where the intervening store may simply fall onto the same cache line.

A.3.4. Progress axiom



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

The progress axiom ensures a minimal forward progress guarantee. It ensures that stores from one hart will eventually be made visible to other harts in the system in a finite amount of time, and that loads from other harts will eventually be able to read those values (or successors thereof). Without this rule, it would be legal, for example, for a spinlock to spin infinitely on a value, even with a store from another hart waiting to unlock the spinlock.

The progress axiom is intended not to impose any other notion of fairness, latency, or quality of service onto the harts in a RISC-V implementation. Any stronger notions of fairness are up to the rest of the ISA and/or up to the platform and/or device to define and implement.

The forward progress axiom will in almost all cases be naturally satisfied by any standard cache coherence protocol. Implementations with non-coherent caches may have to provide some other mechanism to ensure the eventual visibility of all stores (or successors thereof) to all harts.

A.3.5. Overlapping-Address Orderings (Rules 1-3)



Rule 1: b is a store, and a and b access overlapping memory addresses

Rule 2: a and b are loads, x is a byte read by both a and b , there is no store to x between a and b in program order, and a and b return values for x written by different memory operations

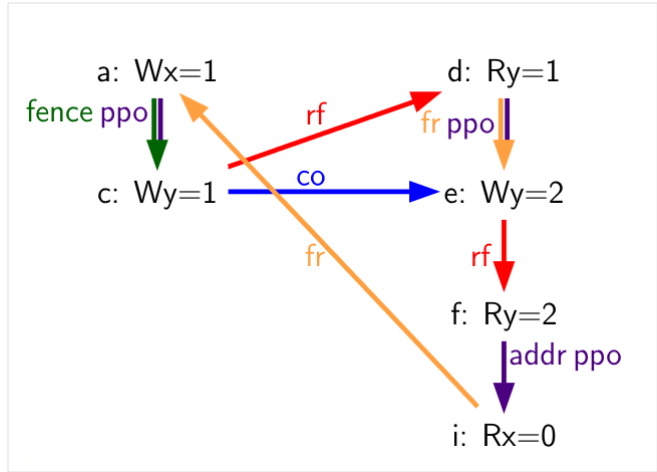
Rule 3: a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model. Same-address orderings from a store to a later load, on the other hand, do not need to be enforced. As discussed in [Load value axiom](#), this reflects the observable behavior of implementations that forward values from buffered stores to later loads.

Same-address load-load ordering requirements are far more subtle. The basic requirement is that a younger load must not return a value that is older than a value returned by an older load in the same hart to the same address. This is often known as "CoRR" (Coherence for Read-Read pairs), or as part of a broader "coherence" or "sequential consistency per location" requirement. Some architectures in the past have relaxed same-address load-load ordering, but in hindsight this is generally considered to complicate the programming model too much, and so RVWMO requires CoRR ordering to be enforced. However, because the global memory order corresponds to the order in which loads perform rather than the ordering of the values being returned, capturing CoRR requirements in terms of the global memory order requires a bit of indirection.

Table 48. 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 48](#), 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:

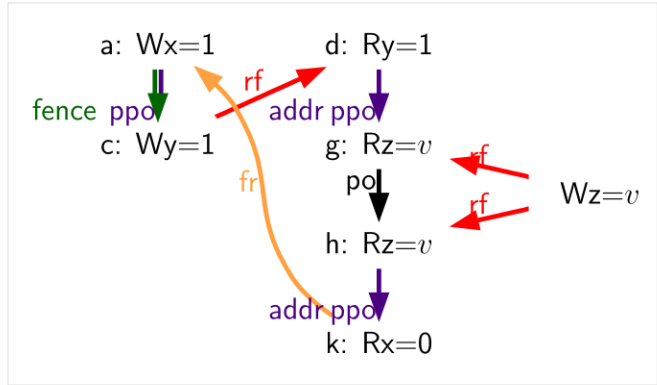
- (d) stalls (for whatever reason; perhaps it's stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer (but does not yet drain to memory)
- (f) executes and forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (a) executes and drains to memory, (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 49. Litmus test RSW (outcome permitted)

Hart 0	Hart 1
li t1, 1	(d) lw a0,0(s1)
(a) sw t1,0(s0)	(e) xor t2,a0,a0
(b) fence w, w	(f) add s4,s2,t2
(c) sw t1,0(s1)	(g) lw a1,0(s4)
	(h) lw a2,0(s2)
	(i) xor t3,a2,a2
	(j) add s0,s0,t3
	(k) lw a3,0(s0)
Outcome: $a0=1$, $a1=v$, $a2=v$, $a3=0$	



Consider the litmus test of Table 49. 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 49 in which $a1$ does not equal $a2$ do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in that case result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, rule 2 forbids this CoRR violation from occurring. As such, rule 2 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

There is one more overlapping-address rule: rule 3 simply states that a value cannot be returned from an AMO or SC to a subsequent load until the AMO or SC has (in the case of the SC, successfully) performed globally. This follows somewhat naturally from the conceptual view that both AMOs and SC instructions are meant to be performed atomically in memory. However, notably, rule 3 states that hardware may not even non-speculatively forward the value being stored by an AMOSWAP to a subsequent load, even though for AMOSWAP that store value is not actually semantically dependent on the previous value in memory, as is the case for the other AMOs. The same holds true even when forwarding from SC store values that are not semantically dependent on the value returned by the paired LR.

The three PPO rules above also apply when the memory accesses in question only overlap partially. This can occur, for example, when accesses of different sizes are used to access the same object. Note also that the base addresses of two overlapping memory operations need not necessarily be the same for two memory accesses to overlap. When misaligned memory accesses are being used, the overlapping-address PPO rules apply to each of the component memory accesses independently.

A.3.6. Fences (Rule 4)



Rule 4: There is a FENCE instruction that orders a before b

By default, the FENCE instruction ensures that all memory accesses from instructions preceding the fence in program order (the "predecessor set") appear earlier in the global memory order than memory accesses from instructions appearing after the fence in program order (the "successor set"). However, fences can optionally further restrict the predecessor set and/or the successor set to a smaller set of memory accesses in order to provide some speedup. Specifically, fences have PR, PW, SR, and SW bits which restrict the predecessor and/or successor sets. The predecessor set includes loads (resp.stores) if and only if PR (resp.PW) is set. Similarly, the successor set includes loads (resp.stores) if and only if SR (resp.SW) is set.

The FENCE encoding currently has nine non-trivial combinations of the four bits PR, PW, SR, and SW, plus one extra encoding FENCE.TSO which facilitates mapping of "acquire+release" or RVTSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, only 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 any other combination of PR, PW, SR, and SW are reserved. We strongly recommend that programmers stick to these six. Other combinations may have unknown or unexpected interactions with the memory model.

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. There is no complex notion of "fence cumulativeness" as found in memory models that are not multi-copy atomic.

A.3.7. Explicit Synchronization (Rules 5-8)



Rule 5: a has an acquire annotation

Rule 6: b has a release annotation

Rule 7: a and b both have RCsc annotations

Rule 8: a is paired with b

An *acquire* operation, as would be used at the start of a critical section, requires all memory operations following the acquire in program order to also follow the acquire in the global memory order. This ensures, for example, that all loads and stores inside the critical section are up to date with respect to the synchronization variable being used to protect it. Acquire ordering can be enforced in one of two ways: with an acquire annotation, which enforces ordering with respect to just the synchronization variable itself, or with a FENCE R,RW, which enforces ordering with respect to all previous loads.

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

```

Example 4. 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

```

Example 5. 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 just as with fences, programmers need not worry about "cumulativity" when analyzing ordering annotations.

A.3.8. Syntactic Dependencies (Rules 9-11)



Rule 9: b has a syntactic address dependency on a

Rule 10: b has a syntactic data dependency on a

Rule 11: b is a store, and b has a syntactic control dependency on a

Dependencies from a load to a later memory operation in the same hart are respected by the RVWMO memory model. The Alpha memory model was notable for choosing *not* to enforce the ordering of such dependencies, but most modern hardware and software memory models consider allowing dependent instructions to be reordered too confusing and counterintuitive. Furthermore, modern code sometimes intentionally uses such dependencies as a particularly lightweight ordering enforcement mechanism.

The terms in [Section 17.1.2](#) 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
```

Example 6. (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 [Example 6](#), (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)
```

Example 7. 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 [Example 7](#), 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)
```

Example 8. 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 [Example 8](#) 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.

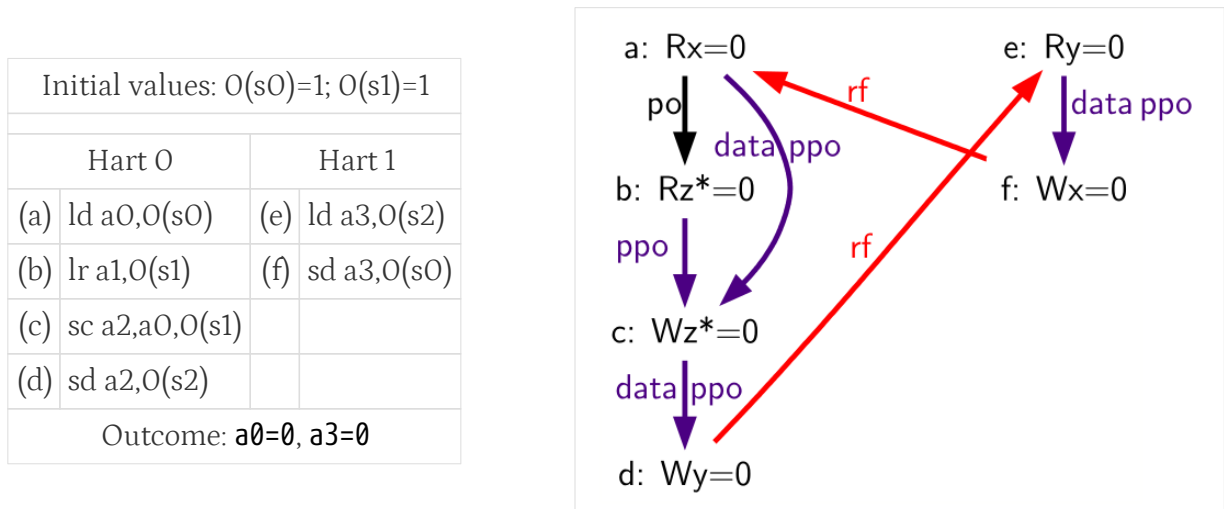
```
1 lw x1,0(x2)
2 bne x1,x0,next
3 next: sw x3,0(x4)
```

Example 9. Another syntactic control dependency

Likewise, consider [Example 9](#). 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 50. 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 50. 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 17.3 does *not* imply that the two **F** instructions must be executed in order. Such a dependency would only serve to ultimately set up later a dependency from both **F** instructions to a later CSR instruction accessing the CSR flag in question.

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



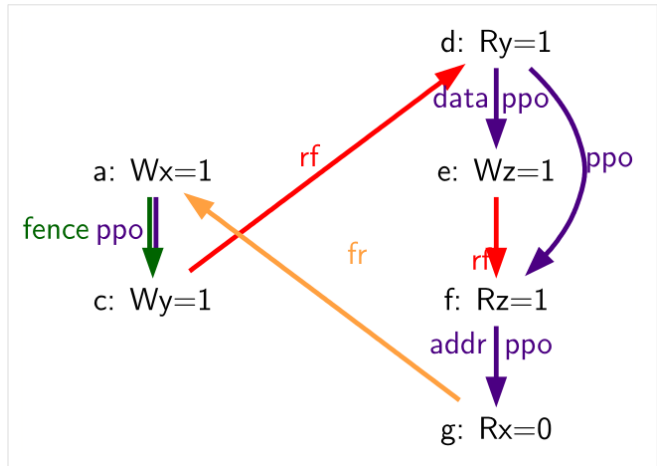
Rule 12: *b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m*

Rule 13: *b is a store, and there exists some instruction m between a and b in program order*

such that m has an address dependency on a

Table 51. 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)

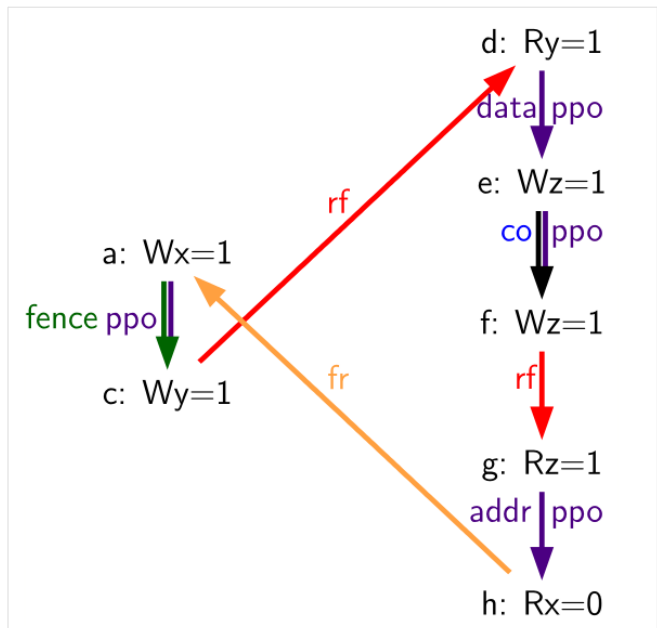
Hart 0	Hart 1
li t1, 1	(d) lw a0, 0(s1)
(a) sw t1,0(s0)	(e) sw a0, 0(s2)
(b) fence w, w	(f) lw a1, 0(s2)
(c) sw t1,0(s1)	xor a2,a1,a1
	add s0,s0,a2
	(g) lw a3,0(s0)
Outcome: a0=1, a3=0	



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 51](#) (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 writeback of (e) before (d) has had a chance to perform. Therefore, (f) will never perform before (d) has performed.

Table 52. 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	



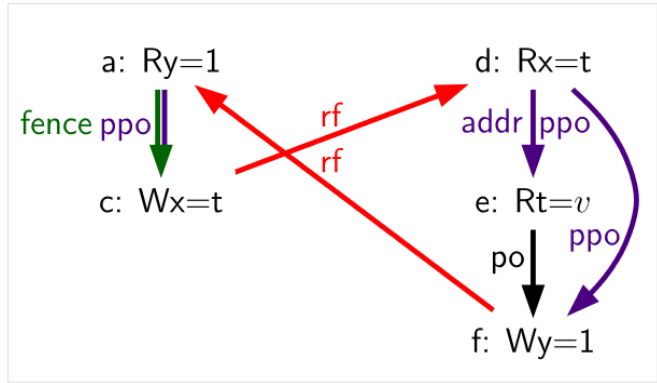
If there were another store to the same address in between (e) and (f), as in [Table 53](#), then (f) would no longer be dependent on the data of (e) being resolved, and hence the dependency of (f) on (d), which produces the data for (e), would be broken.

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 53. 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 53 (f) cannot be executed until the address for (e) is resolved, because it may turn out that the addresses match; i.e., that **a1=s0**. Therefore, (f) cannot be sent to memory before (d) has executed and confirmed whether the addresses do indeed overlap.

A.4. Beyond Main Memory

RVWMO does not currently attempt to formally describe how FENCE.I, SFENCE.VMA, I/O fences, and PMAs behave. All of these behaviors will be described by future formalizations. In the meantime, the behavior of FENCE.I is described in Chapter 3, the behavior of SFENCE.VMA is described in the RISC-V Instruction Set Privileged Architecture Manual, and the behavior of I/O fences and the effects of PMAs are described below.

A.4.1. Coherence and Cacheability

The RISC-V Privileged ISA defines Physical Memory Attributes (PMAs) which specify, among other things, whether portions of the address space are coherent and/or cacheable. See the RISC-V Privileged ISA Specification for the complete details. Here, we simply discuss how the various details in each PMA relate to the memory model:

- Main memory vs.I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.
- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Cacheability PMAs: the cacheability PMAs in general do not affect the memory model. Non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless. However, some platform-specific and/or device-specific cacheability settings may differ.
- Coherence PMAs: The memory consistency model for memory regions marked as non-coherent in PMAs is currently platform-specific and/or device-specific: the load-value axiom, the atomicity axiom, and the progress axiom all may be violated with non-coherent memory. Note however that coherent memory does not require a hardware cache coherence protocol. The RISC-V Privileged

ISA Specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise.

- **Idempotency PMAs:** Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

A.4.2. I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects and may return values other than the value "written" by the most recent store to the same address. Nevertheless, the following preserved program order rules still generally apply for accesses to I/O memory: memory access a precedes memory access b in global memory order if a precedes b in program order and one or more of the following holds:

1. a precedes b in preserved program order as defined in [Chapter 17](#), 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)
```

Example 10. 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 [Example 10](#), suppose $0(a0)$ is in main memory and $0(a1)$ is the address of a device register in I/O memory. If the device accesses $0(a0)$ upon receiving the MMIO write, then that load must conceptually appear after the first store to $0(a0)$ according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and main memory

accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

Many architectures include separate notions of "ordering" and "completion" fences, especially as it relates to I/O (as opposed to regular main memory). Ordering fences simply ensure that memory operations stay in order, while completion fences ensure that predecessor accesses have all completed before any successors are made visible. RISC-V does not explicitly distinguish between ordering and completion fences. Instead, this distinction is simply inferred from different uses of the FENCE bits.

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices and DMA operations are required to access memory coherently and via strongly ordered I/O channels. Therefore, accesses to regular main memory regions that are concurrently accessed by external devices can also use the standard synchronization mechanisms. Implementations that do not conform to the Unix Platform Specification and/or in which devices do not access memory coherently will need to use mechanisms (which are currently platform-specific or device-specific) to enforce coherency.

I/O regions in the address space should be considered non-cacheable regions in the PMAs for those regions. Such regions can be considered coherent by the PMA if they are not cached by any agent.

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe) may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

A.5. Code Porting and Mapping Guidelines

Table 54. 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}.aqr1 OR loop:l.r.{w d}.aq; <op>; sc.{w d}.aqr1; bnez loop</code>
Fence	<code>fence rw,rw</code>

Table 54 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 54 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 55. 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 55 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 56. 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}.aqr1</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}.r1</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 56 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 56 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 57. 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}.rl</code>
<code>atomic <op></code>	<code>amo <op>.{w d}.aqr1</code>
Linux Construct	RVWMO LR/SC Mapping
<code>atomic <op> relaxed</code>	<code>loop:lr.{w d}; <op>; sc.{w d}; bnez loop</code>
<code>atomic <op> acquire</code>	<code>loop:lr.{w d}.aq; <op>; sc.{w d}; bnez loop</code>
<code>atomic <op> release</code>	<code>loop:lr.{w d}; <op>; sc.{w d}.aqr1^*; bnez loop OR</code>
	<code>fence.tso; loop:lr.{w d}; <op >; sc.{w d}^*; bnez loop</code>
<code>atomic <op></code>	<code>loop:lr.{w d}.aq; <op>; sc.{w d}.aqr1; bnez loop</code>

With regards to [Table 57](#), 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 57](#) 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 57](#).

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 [Example 11](#). 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) amoswap.d.aq a1,t1,0(s1)
16 bnez a1,loop
17 (e) lw a2,0(s2)

```

Example 11. Orderings between critical sections in Linux

[Table 58](#) 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 59](#) 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 58](#) sequentially consistent store, followed by a [Table 59](#) sequentially consistent load can be reordered unless the [Table 58](#) mapping of stores is strengthened by either adding a second fence or mapping the store to `amoswap.rl` instead.

Table 58. 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>

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

Table 59. 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

C/C++ Construct	RVWMO Mapping
<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}.aqrl</code>
<code>atomic_<op>(memory_order_seq_cst)</code>	<code>amo<op>.{w d}.aqrl</code>
C/C++ Construct	RVWMO LR/SC Mapping
<code>atomic_<op>(memory_order_relaxed)</code>	<code>lr.{w d}; <op>; sc.{w d}</code>
<code>atomic_<op>(memory_order_acquire)</code>	<code>lr.{w d}.aq; <op>; sc.{w d}</code>
<code>atomic_<op>(memory_order_release)</code>	<code>lr.{w d}; <op>; sc.{w d}.rl</code>
<code>atomic_<op>(memory_order_acq_rel)</code>	<code>lr.{w d}.aq; <op>; sc.{w d}.rl</code>
<code>atomic_<op>(memory_order_seq_cst)</code>	<code>lr.{w d}.aq* <op>; sc.{w d}.rl</code>
* must be <code>lr.{w d}.aqrl</code> in order to interoperate with code mapped per Table 58	

Any AMO can be emulated by an LR/SC pair, but care must be taken to ensure that any PPO orderings that originate from the LR are also made to originate from the SC, and that any PPO orderings that terminate at the SC are also made to terminate at the LR. For example, the LR must also be made to respect any data dependencies that the AMO has, given that load operations do not otherwise have any notion of a data dependency. Likewise, the effect a FENCE R,R elsewhere in the same hart must also be made to apply to the SC, which would not otherwise respect that fence. The emulator may achieve this effect by simply mapping AMOs onto `lr.aq; <op>; sc.aqrl`, matching the mapping used elsewhere for fully ordered atomics.

These C11/C++11 mappings require the platform to provide the following Physical Memory Attributes (as defined in the RISC-V Privileged ISA) for all memory:

- main memory
- coherent
- AMOArithmetic
- RsrVEventual

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

A.6. Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design only admits executions that satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or *other-multi-copy-atomic*): any store value that is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no early inter-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer (unless of course it is done in such a way that no new illegal behaviors become architecturally visible). Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1 and 4-8 regularly and actively.
- expert programmers will use PPO rules 9-11 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 2-3 and 12-13 directly. These are included to facilitate common microarchitectural optimizations (rule 2) and the operational formal modeling approach (rules 3 and 12-13) described in Section B.3. They also facilitate the process of porting code from other architectures that have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1 and 3-6 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 2 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.
- PPO rule 7 may not be immediately obvious to architects, but it is a standard memory model requirement
- The load value axiom, the atomicity axiom, and PPO rules 8-13 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not **optimized away**.

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), regardless of the bits actually set
- implement all fences with PW and SR as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), as PW with SR is the most expensive of the four possible main memory ordering components anyway
- emulate *aq* and *rl* as described in [Section A.5](#)
- enforcing all same-address load-load ordering, even in the presence of patterns such as **fri-rfi** and **RSW**
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or LR to the same address
- forbid any forwarding of a value from an AMO or SC in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any main memory fences that do not include PW and SR ordering (e.g., as Ztso implementations will do)
- implement all atomics to be RCsc or even fully ordered, regardless of annotation

Architectures that implement RVTSO can safely do the following:

- Ignore all fences that do not have both PW and SR (unless the fence also orders I/O)
- Ignore all PPO rules except for rules 4 through 7, since the rest are redundant with other PPO rules under RVTSO assumptions

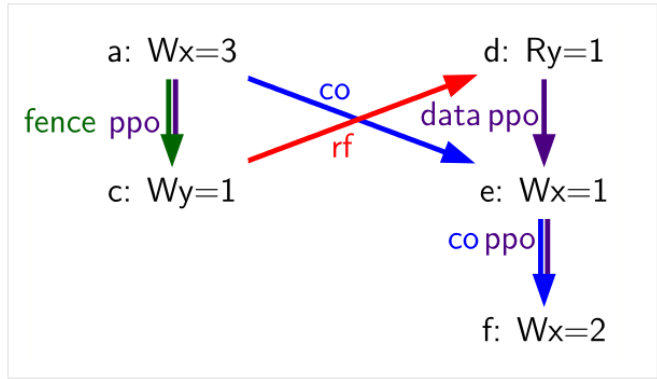
Other general notes:

- Silent stores (i.e., stores that write the same value that already exists at a memory location) behave like any other store from a memory model point of view. Likewise, AMOs which do not actually change the value in memory (e.g., an AMOMAX for which the value in *rs2* is smaller than the value currently in memory) are still semantically considered store operations. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in cases such as RSW [Section A.3.5](#) which tend to be incompatible with silent stores.
- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

Table 60. 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 2
- (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 7
- (e) precedes (f) in the global memory order because of rule 1

In other words the final value of the memory location whose address is in **s0** must be 2 (the value written by the store (f)) and cannot be 3 (the value written by the store (a)).

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

A.6.1. Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- "V" vector ISA extensions
- "J" JIT extension
- Native encodings for load and store opcodes with *aq* and *rl* set
- Fences limited to certain addresses
- Cache writeback/flush/invalidate/etc.instructions

A.7. Known Issues

A.7.1. Mixed-size RSW

Table 61. 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 62. 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 63. 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

Hart 0		Hart 1	
			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 61-Table 63](#). 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 **(p-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 B: Formal Memory Model Specifications, Version 0.1

To facilitate formal analysis of RVWMO, this chapter presents a set of formalizations using different tools and modeling approaches. Any discrepancies are unintended; the expectation is that the models describe exactly the same sets of legal behaviors.

This appendix should be treated as commentary; all normative material is provided in [Chapter 17](#) and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in [Section A.7](#). Any other discrepancies are unintentional.

B.1. Formal Axiomatic Specification in Alloy

We present a formal specification of the RVWMO memory model in Alloy (alloy.mit.edu). This model is available online at github.com/daniellustig/riscv-memory-model.

The online material also contains some litmus tests and some examples of how Alloy can be used to model check some of the mappings in [Section A.5](#).

```
// =RVWMO PPO=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc := Store
  + rdw
  + (AMO + StoreConditional) <: rfi

  // explicit synchronization
  + ppo_fence
  + Acquire <: ^po := MemoryEvent
  + MemoryEvent <: ^po := Release
  + RCsc <: ^po := RCsc
  + pair

  // syntactic dependencies
  + addrdep
  + datadep
  + ctrldep := Store

  // pipeline dependencies
  + (addrdep+datadep).rfi
  + addrdep.^po := Store
}

// the global memory order respects preserved program order
fact { ppo in ^gmo }
```

Example 12. 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 RISC_V_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) }

```

Example 13. The RVWMO memory model formalized in Alloy (3/5: model of memory)

```

// =Basic model rules=

// Ordering annotation groups
fun Acquire : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.acquireRCsc }
fun Release : MemoryEvent { MemoryEvent.releaseRCpc + MemoryEvent.releaseRCsc }
fun RCpc : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.releaseRCpc }
fun RCsc : MemoryEvent { MemoryEvent.acquireRCsc + MemoryEvent.releaseRCsc }

// There is no such thing as store-acquire or load-release, unless it's both
fact { Load & Release in Acquire }
fact { Store & Acquire in Release }

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
  (Load <: ^po := FencePRSR).(^po := Load)
  + (Load <: ^po := FencePRSW).(^po := Store)
  + (Store <: ^po := FencePWSR).(^po := Load)
  + (Store <: ^po := FencePWSW).(^po := Store)
  + (Load <: ^po := FenceTSO).(^po := MemoryEvent)
  + (Store <: ^po := FenceTSO).(^po := Store)
}

```

```

}

// auxiliary definitions
fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// initial stores
fun NonInit : set Event { Hart.start.*po }
fun Init : set Event { Event - NonInit }
fact { Init in StoreNormal }
fact { Init->(MemoryEvent & NonInit) in ^gmo }
fact { all e: NonInit | one e.*~po.~start } // each event is in exactly one hart
fact { all a: Address | one Init & a.~address } // one init store per address
fact { no Init <: po and no po := Init }

```

Example 14. The RVWMO memory model formalized in Alloy (4/5: Basic model rules)

```

// po
fact { acyclic[po] }

// gmo
fact { total[^gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf.~rf in iden } // each read returns the value of only one write
fact { rf in Store <: address.~address := Load }
fun rfi : MemoryEvent->MemoryEvent { rf & (*po + *~po) }

//dep
fact { no StoreNormal <: (addrdep + ctrldep + datadep) }
fact { addrdep + ctrldep + datadep + pair in ^po }
fact { datadep in datadep := Store }
fact { ctrldep.*po in ctrldep }
fact { no pair & (^po := (LoadReserve + StoreConditional)).^po }
fact { StoreConditional in LoadReserve.pair } // assume all SCs succeed

// rdw
fun rdw : Event->Event {
  (Load <: po_loc := Load) // start with all same_address load-load pairs,
  - (~rf.rf) // subtract pairs that read from the same store,
  - (po_loc.rfi) // and subtract out "fri-rfi" patterns
}

// filter out redundant instances and/or visualizations
fact { no gmo & gmo.gmo } // keep the visualization uncluttered
fact { all a: Address | some a.~address }

// =Optional: opcode encoding restrictions=

// the list of blessed fences
fact { Fence in
  Fence.pr.sr
  + Fence.pw.sw

```

```

+ Fence.pr.pw.sw
+ Fence.pr.sr.sw
+ FenceTSO
+ Fence.pr.pw.sr.sw
}

pred restrict_to_current_encodings {
  no (LoadNormal + StoreNormal) & (Acquire + Release)
}

// =Alloy shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
  all disj e, e': bag | e->e' in rel + ~rel
  acyclic[rel]
}

```

Example 15. The RVWMO memory model formalized in Alloy (5/5: Auxiliaries)

B.2. Formal Axiomatic Specification in Herd

The tool **herd** takes a memory model and a litmus test as input and simulates the execution of the test on top of the memory model. Memory models are written in the domain specific language Cat. This section provides two Cat memory model of RVWMO. The first model, [Example 17](#), follows the *global memory order*, Chapter [Chapter 17](#), definition of RVWMO, as much as is possible for a Cat model. The second model, [Example 18](#), 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

```

Example 16. *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

```

Example 17. 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

Example 18. `riscv.cat`, an alternative herd presentation of the RVWMO memory model (3/3)

B.3. An Operational Memory Model

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behavior as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavor: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from more implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behavior of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

The operational model, together with a fragment of the RISC-V ISA semantics (RV64I and A), are integrated into the `rmem` exploration tool (github.com/rem-s-project/rmem). `rmem` can explore litmus tests (see [Section A.2](#)) and small ELF binaries exhaustively, pseudo-randomly and interactively. In `rmem`, the ISA semantics is expressed explicitly in Sail (see github.com/rem-s-project/sail for the Sail language, and github.com/rem-s-project/sail-riscv for the RISC-V ISA model), and the concurrency semantics is expressed in Lem (see github.com/rem-s-project/lem for the Lem language).

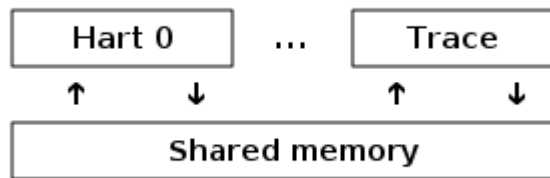
`rmem` has a command-line interface and a web-interface. The web-interface runs entirely on the client side, and is provided online together with a library of litmus tests: www.cl.cam.ac.uk/. The command-line interface is faster than the web-interface, specially in exhaustive mode.

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 un-taken alternative paths are discarded.

Each instruction instance in the instruction tree has a state that includes an execution state of the intra-instruction semantics (the ISA pseudocode for this instruction). The model uses a formalization of the intra-instruction semantics in Sail. One can think of the execution state of an instruction as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information about the instance's memory and register footprints, its register reads and writes, its memory operations, whether it is finished, etc.

Model transitions

The model defines, for any model state, the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Execution of a single instruction will typically involve many transitions, and they may be interleaved in operational-model execution with transitions arising from other instructions. Each transition arises from a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its hart state and the shared memory state, but it does not depend on other hart states, and it will not change them. The transitions are introduced below and defined in [Section B.3.5](#), with a precondition and a construction of the post-transition model state for each.

Transitions for all instructions:

- [Fetch instruction](#): This transition represents a fetch and decode of a new instruction instance, as a program order successor of a previously fetched instruction instance (or the initial fetch address).

The model assumes the instruction memory is fixed; it does not describe the behavior of self-modifying code. In particular, the [Fetch instruction](#) transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.

- [Register write](#): This is a write of a register value.
- [Register read](#): This is a read of a register value from the most recent program-order-predecessor instruction instance that writes to that register.
- [Pseudocode internal step](#): This covers pseudocode internal computation: arithmetic, function calls,

etc.

- **Finish instruction**: At this point the instruction pseudocode is done, the instruction cannot be restarted, memory accesses cannot be discarded, and all memory effects have taken place. For conditional branch and indirect jump instructions, any program order successors that were fetched from an address that is not the one that was written to the *pc* register are discarded, together with the sub-tree of instruction instances below them.

Transitions specific to load instructions:

- **Initiate memory load operations**: At this point the memory footprint of the load instruction is provisionally known (it could change if earlier instructions are restarted) and its individual memory load operations can start being satisfied.
- **Satisfy memory load operation by forwarding from 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 ◦ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behavior; the • cannot. Although [Fetch instruction](#) is marked with a •, it can be taken eagerly as long as it is not taken infinitely many times.

An instance of a non-AMO load instruction, after being fetched, will typically experience the following transitions in this order:

1. [Register read](#)
2. [Initiate memory load operations](#)
3. [Satisfy memory load operation by forwarding from unpropagated stores](#) and/or [Satisfy memory load operation from memory](#) (as many as needed to satisfy all the load operations of the instance)
4. [Complete load operations](#)
5. [Register write](#)
6. [Finish instruction](#)

Before, between and after the transitions above, any number of [Pseudocode internal step](#) transitions may appear. In addition, a [Fetch instruction](#) transition for fetching the instruction in the next program location will be available until it is taken.

This concludes the informal description of the operational model. The following sections describe the formal operational model.

B.3.1. Intra-instruction Pseudocode Execution

The intra-instruction semantics for each instruction instance is expressed as a state machine, essentially running the instruction pseudocode. Given a pseudocode execution state, it computes the next state. Most states identify a pending memory or register operation, requested by the pseudocode, which the memory model has to do. The states are (this is a tagged union; tags in small-caps):

<code>Load_mem(kind, address, size, load_continuation)</code>	- memory load operation
<code>Early_sc_fail(res_continuation)</code>	- allow <code>sc</code> to fail early
<code>Store_ea(kind, address, size, next_state)</code>	- memory store effective address
<code>Store_memv(mem_value, store_continuation)</code>	- memory store value
<code>Fence(kind, next_state)</code>	- fence
<code>Read_reg(reg_name, read_continuation)</code>	- register read
<code>Write_reg(reg_name, reg_value, next_state)</code>	- register write

Internal(<i>next_state</i>)	- pseudocode internal step
Done	- end of pseudocode

Here:

- *mem_value* and *reg_value* are lists of bytes;
- *address* is an integer of XLEN bits;

for load/store, *kind* identifies whether it is **lr/sc**, acquire-RCpc/release-RCpc, acquire-RCsc/release-RCsc, acquire-release-RCsc; * for fence, *kind* identifies whether it is a normal or TSO, and (for normal fences) the predecessor and successor ordering bits; * *reg_name* identifies a register and a slice thereof (start and end bit indices); and the continuations describe how the instruction instance will continue for each value that might be provided by the surrounding memory model (the *load_continuation* and *read_continuation* take the value loaded from memory and read from the previous register write, the *store_continuation* takes *false* for an **sc** that failed and *true* in all other cases, and *res_continuation* takes *false* if the **sc** fails and *true* otherwise).



*For example, given the load instruction **lw x1,0(x2)**, an execution will typically go as follows. The initial execution state will be computed from the pseudocode for the given opcode. This can be expected to be *Read_reg(x2, read_continuation)*. Feeding the most recently written value of register **x2** (the instruction semantics will be blocked if necessary until the register value is available), say **0x4000**, to *read_continuation* returns *Load_mem(plain_load, 0x4000, 4, load_continuation)*. Feeding the 4-byte value loaded from memory location **0x4000**, say **0x42**, to *load_continuation* returns *Write_reg(x1, 0x42, Done)*. Many *Internal(next_state)* states may appear before and between the states above.*

Notice that writing to memory is split into two steps, *Store_ea* and *Store_memv*: the first one makes the memory footprint of the store provisionally known, and the second one adds the value to be stored. We ensure these are paired in the pseudocode (*Store_ea* followed by *Store_memv*), but there may be other steps between them.



*It is observable that the *Store_ea* can occur before the value to be stored is determined. For example, for the litmus test **LB+fence.r.rw+data-po** to be allowed by the operational model (as it is by RVWMO), the first store in Hart 1 has to take the *Store_ea* step before its value is determined, so that the second store can see it is to a non-overlapping memory footprint, allowing the second store to be committed out of order without violating coherence.*

The pseudocode of each instruction performs at most one store or one load, except for AMOs that perform exactly one load and one store. Those memory accesses are then split apart into the architecturally atomic units by the hart semantics (see [Initiate memory load operations](#) and [Initiate memory store operation footprints](#) below).

Informally, each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit (or from the hart's initial register state if there is no such write). Hence, it is essential to know the register write footprint of each instruction instance, which we calculate when the instruction instance is created (see the [Fetch instruction](#) action of below). We ensure in the pseudocode that each instruction does at most one register write to each register bit, and also that it does not try to read a register value it just wrote.

Data-flow dependencies (address and data) in the model emerge from the fact that each register read

has to wait for the appropriate register write to be executed (as described above).

B.3.2. Instruction Instance State

Each instruction instance i has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, AMO, fence, branch/jump or a **simple** instruction (this also includes a *kind* similar to the one described for the pseudocode execution states);
- *src_regs*, the set of source *_reg_name_s* (including system registers), as statically determined from the pseudocode of the instruction;
- *dst_regs*, the destination *_reg_name_s* (including system registers), as statically determined from the pseudocode of the instruction;
- *pseudocode_state* (or sometimes just **state** for short), one of (this is a tagged union; tags in small-caps):

Plain(<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.*

B.3.3. Hart State

The model state of a single hart comprises:

- *hart_id*, a unique identifier of the hart;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial instruction fetch address;
- *instruction_tree*, a tree of the instruction instances that have been fetched (and not discarded), in program order.

B.3.4. Shared Memory State

The model state of the shared memory comprises a list of memory store operations, in the order they propagated to the shared memory.

When a store operation is propagated to the shared memory it is simply added to the end of the list. When a load operation is satisfied from memory, for each byte of the load operation, the most recent corresponding store slice is returned.



*For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to memory store operation slices, where each memory location is mapped to a one-byte slice of the most recent memory store operation to that location. However, this abstraction is not detailed enough to properly handle the **sc** instruction. The RVWMO allows store operations from the same hart as the **sc** to intervene between the store operation of the **sc** and the store operations the paired **lr** read from. To allow such store operations to intervene, and forbid others, the array abstraction must be extended to record more information. Here, we use a list as it is very simple, but a more efficient and scalable implementations should probably use something better.*

B.3.5. Transitions

Each of the paragraphs below describes a single kind of system transition. The description starts with a condition over the current system state. The transition can be taken in the current state only if the condition is satisfied. The condition is followed by an action that is applied to that state when the transition is taken, in order to generate the new system state.

Fetch instruction

A possible program-order-successor of instruction instance i can be fetched from address loc if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the hart's *instruction_tree* are from loc ; and
2. if i 's pseudocode has already written an address to pc , then loc must be that address, otherwise loc is:
 - for a conditional branch, the successor address or the branch target address;
 - for a (direct) jump and link instruction (**jal**), the target address;
 - for an indirect jump instruction (**jalr**), any address; and
 - for any other instruction, $i.program_loc + 4$.

Action: construct a freshly initialized instruction instance i' for the instruction in the program memory at loc , with state `Plain(isa_state)`, computed from the instruction pseudocode, including the static information available from the pseudocode such as its *instruction_kind*, *src_regs*, and *dst_regs*, and add i' to the hart's *instruction_tree* as a successor of i .

The possible next fetch addresses (loc) are available immediately after fetching i and the model does not need to wait for the pseudocode to write to pc ; this allows out-of-order execution, and speculation past conditional branches and jumps. For most instructions these addresses are easily obtained from the instruction pseudocode. The only exception to that is the indirect jump instruction (**jalr**), where the address depends on the value held in a register. In principle the mathematical model should allow speculation to arbitrary addresses here. The exhaustive search in the **rmem** tool handles this by running the exhaustive search multiple times with a growing set of possible next fetch addresses for each indirect jump. The initial search uses empty sets, hence there is no fetch after indirect jump instruction until the pseudocode of the instruction writes to pc , and then we use that value for fetching the next instruction. Before starting the next iteration of exhaustive search, we collect for each indirect jump (grouped by code location) the set of values it wrote to pc in all the executions in the previous search iteration, and use that as possible next fetch addresses of the instruction. This process terminates when no new fetch addresses are detected.

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 17.1.1](#) 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.

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.

Satisfy memory load operation from memory

For an instruction instance i of a non-AMO load instruction or an AMO instruction in the context of the [Satisfy, commit and propagate operations of an AMO](#) transition, any memory load operation mlo in $i.mem_loads$ that has unsatisfied slices, can be satisfied from memory if all the conditions of `<sat_by_forwarding, Satisfy memory load operation by forwarding from unpropagated stores>` are satisfied. Action: let $msoss$ be the memory store operation slices from memory covering the unsatisfied slices of mlo , and apply the action of [Satisfy memory operation by forwarding from unpropagates stores](#).



Note that [Satisfy memory operation by forwarding from unpropagates stores](#) might leave some slices of the memory load operation unsatisfied, those will have to be satisfied by taking the transition again, or taking [Satisfy memory load operation from memory](#). [Satisfy memory load operation from memory](#), on the other hand, will always satisfy all the unsatisfied slices of the memory load operation.

Complete load operations

A load instruction instance i in state $\text{Pending_mem_loads}(\text{load_continuation})$ can be completed (not to be confused with finished) if all the memory load operations $i.\text{mem_loads}$ are entirely satisfied (i.e. there are no unsatisfied slices). Action: update the state of i to $\text{Plain}(\text{load_continuation}(\text{mem_value}))$, where mem_value is assembled from all the memory store operation slices that satisfied $i.\text{mem_loads}$.

Early sc fail

An **sc** instruction instance i in state $\text{Plain}(\text{Early_sc_fail}(\text{res_continuation}))$ can always be made to fail. Action: update the state of i to $\text{Plain}(\text{res_continuation}(\text{false}))$.

Paired sc

An **sc** instruction instance i in state $\text{Plain}(\text{Early_sc_fail}(\text{res_continuation}))$ can continue its (potentially successful) execution if i is paired with an **lr**. Action: update the state of i to $\text{Plain}(\text{res_continuation}(\text{true}))$.

Initiate memory store operation footprints

An instruction instance i in state $\text{Plain}(\text{Store_ea}(\text{kind}, \text{address}, \text{size}, \text{next_state}))$ 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 $\text{address} \dots \text{address} + \text{size} - 1$.
2. set $i.\text{mem_stores}$ to msos ; and
3. update the state of i to $\text{Plain}(\text{next_state})$.

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).

Instantiate memory store operation values

An instruction instance i in state `Plain(Store_memv(mem_value, store_continuation))` can always instantiate the values of the memory store operations $i.mem_stores$. Action:

1. split mem_value between the memory store operations $i.mem_stores$; and
2. update the state of i to `Pending_mem_stores(store_continuation)`.

Commit store instruction

An uncommitted instruction instance i of a non-**sc** store instruction or an **sc** instruction in the context of the [Commit and propagate store operation of an sc](#) transition, in state `Pending_mem_stores(store_continuation)`, can be committed (not to be confused with propagated) if:

1. i has fully determined data;
2. all program-order-previous conditional branch and indirect jump instructions are finished;
3. all program-order-previous **fence** instructions with **.sw** set are finished;
4. all program-order-previous **fence.tso** instructions are finished;
5. all program-order-previous load-acquire instructions are finished;
6. all program-order-previous store-acquire-release instructions are finished;
7. if i is a store-release, all program-order-previous instructions are finished;
8. all program-order-previous memory access instructions have a fully determined memory footprint;
9. all program-order-previous store instructions, except for **sc** that failed, have initiated and so have non-empty mem_stores ; and
10. all program-order-previous load instructions have initiated and so have non-empty mem_loads .

Action: record that i is committed.



Notice that if condition 8 is satisfied the conditions 9 and 10 are also satisfied, or will be satisfied after taking some eager transitions. Hence, requiring them does not strengthen the model. By requiring them, we guarantee that previous memory access instructions have taken enough transitions to make their memory operations visible for the condition check of, which is the next transition the instruction will take, making that condition simpler.

Propagate store operation

For a committed instruction instance i in state `Pending_mem_stores(store_continuation)`, and an unpropagated memory store operation mso in $i.mem_stores$, mso can be propagated if:

1. all memory store operations of program-order-previous store instructions that overlap with mso have already propagated;
2. all memory load operations of program-order-previous load instructions that overlap with mso have already been satisfied, and (the load instructions) are *non-restartable* (see definition below); and

3. all memory load operations that were satisfied by forwarding *mso* are entirely satisfied.

Where a non-finished instruction instance *j* is *non-restartable* if:

1. there does not exist a store instruction *s* and an unpropagated memory store operation *mso* of *s* such that applying the action of the [Propagate store operation](#) transition to *mso* will result in the restart of *j*; and
2. there does not exist a non-finished load instruction *l* and a memory load operation *mlo* of *l* such that applying the action of the [Satisfy memory load operation by forwarding from unpropagated stores](#)/[Satisfy memory load operation from memory](#) transition (even if *mlo* is already satisfied) to *mlo* will result in the restart of *j*.

Action:

1. update the shared memory state with *mso*;
2. update *i.mem_stores* to indicate that *mso* was propagated; and
3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction *i'* program-order-after *i* and every memory load operation *mlo'* of *i'* that was satisfied from *msoss'*, if there exists a memory store operation slice *msos'* in *msoss'* that overlaps with *mso* and is not from *mso*, and *msos'* is not from a program-order-successor of *i*, restart *i'* and its *restart-dependents* (see [Satisfy memory load operation by forwarding from unpropagated stores](#)).

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](#).

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 [Eaarly sc fail](#) transition instead of waiting for this transition.

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))`.

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. [Compleare 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.

Commit fence

A fence instruction instance i in state $\text{Plain}(\text{Fence}(\text{kind}, \text{next_state}))$ 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 $\text{Plain}(\text{next_state})$.

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 $\text{Plain}(\text{read_cont}(\text{reg_value}))$.

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.

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 $\text{Plain}(\text{next_state})$.

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 $\overline{\text{cfp}}$ be the complement of cfp in the memory footprint of i . If $\overline{\text{cfp}}$ is not empty:
 - i. i' has a fully determined memory footprint;
 - ii. i' has no unpropagated memory store operations that overlap with $\overline{\text{cfp}}$; and
 - iii. if i' is a load with a memory footprint that overlaps with $\overline{\text{cfp}}$, then all the memory load operations of i' that overlap with $\overline{\text{cfp}}$ are satisfied and i' is *non-restartable* (see the [Propagate store operation](#) transition for how to determined if an instruction is non-restartable).

Here, a memory store operation is called *fixed* if the store instruction has fully determined data.

2. i has a fully determined data; and
3. if i is not a fence, all program-order-previous conditional branch and indirect jump instructions are finished.

Action:

1. if i is a conditional branch or indirect jump instruction, discard any untaken paths of execution, i.e., remove all instruction instances that are not reachable by the branch/jump taken in *instruction_tree*; and
2. record the instruction as finished, i.e., set *finished* to *true*.

B.3.6. Limitations

- The model covers user-level RV64I and RV64A. In particular, it does not support the misaligned atomicity granule PMA or the total store ordering extension "Ztso". It should be trivial to adapt the model to RV32I/A and to the G, Q and C extensions, but we have never tried it. This will involve, mostly, writing Sail code for the instructions, with minimal, if any, changes to the concurrency model.
- The model covers only normal memory accesses (it does not handle I/O accesses).
- The model does not cover TLB-related effects.
- The model assumes the instruction memory is fixed. In particular, the [Fetch instruction](#) transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.
- The model does not cover exceptions, traps and interrupts.

DRAFT

Index

@

(calling convention standard), 109
 (compressed formats), 109
 (compressed. C.BREAKPOINTINSTR), 119
 (compressed. C.CA), 118
 (compressed. C.CR), 117
 (compressed. C.DIINST), 119
 (compressed. C.NOPINSTR), 119
 (register source spcifiers c-ext), 110

A

atomics
 misaligned, 95

B

bi-endian, 18

C

compressed
 C.ANDI, 117
 C.SRLI
 C.SRAI, 117
 cb-format load and store, 115
 CI, 116
 CIW, 116
 cj-format load and store, 114
 cr-format load and store, 114
 cs-format load and store, 113
 integer constant generation, 115
 integer register-immediate, 116
 register-based load and store, 113

core

accelerator, 11
 cluster
 multiprocessors, 11
 component, 11
 extensions
 coprocessor, 11

D

decomposition, 96
 design
 high performance, 95
 scalable, 95
 double-precision
 floating point, 83

to single-precision, 85

E

endian
 bi-, 18
 little and big, 18
 exceptions, 19

F

FENCE, 95
 FENCE.I
 finer-grained, 40
 forward compatibility, 40
 synchronization, 40
 floating point
 convert and move, 85
 double precision, 83
 fused multiply-add, 79
 load and store, 84

floating-point
 classification, 82
 classify, 86
 compare, 86
 conversion, 80
 exception flag, 76
 requirements, 83
 supported precisions, 83

H

hart
 execution environment, 12
 HINT
 PAUSE, 45

I

ILEN, 16
 IMAFD, 17
 instruction length encoding, 17
 interrupts, 19
 ISA
 definition, 10

M

memory access
 implicit and explicit, 15, 16
 MUL
 DIV, 54
 div by zero, 55
 DIVU, 54

MULH, [54](#)
MULHSU, [54](#)
MULHU, [54](#)
Zmmul, [56](#)

N

NaN

generation, [77](#)
propagation, [77](#)

O

operations

memory, [96](#)
subnormal, [77](#)

P

PAUSE

duration, [45](#)
encoding, [45](#)
energy consumption, [45](#)
HINT, [45](#)
LR/RC sequences, [45](#)

R

RV128

design, [52](#)
evolution, [52](#)

RV128I

as relates to RV64I, [52](#)
compatibility with RV64, [52](#)
LOU, [52](#)

RV32E

design, [46](#)
difference from RV32I, [46](#)

RV64I

compares, [47](#)
HINT, [49](#)
LD, [49](#)
LUI, [48](#)
RV64I-only, [48](#)
shifts, [47](#)
SLLI, [48](#)
SRKIW, [48](#)
SRLIW, [48](#)

RV64I-only

ADDW, [49](#)
SLLW, [49](#)
SRAW, [49](#)
SRLW, [49](#)
SUBW, [49](#)

RVWMO, [95](#)

S

SFENCE, [95](#)

single-precision

to double-precision, [85](#)

store instruction word

not included, [39](#)

T

tininess

handling, [77](#)

traps, [19](#)

U

unspecified

behaviors, [20](#)
values, [20](#)

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.

IEEE Standard for a 32-bit microprocessor. (1994). IEEE Std. 1754-1994.

ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic. (2008). "Institute of Electrical and Electronic Engineers".

Amdahl, G. M., Blaauw, G. A., & F. P. Brooks, J. (1964). Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2).

Buchholz, W. (1962). *Planning a computer system: Project Stretch*. McGraw-Hill Book Company.

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.

Heil, T. H., & Smith, J. E. (1996). *Selective Dual Path Execution*. University of Wisconsin - Madison.

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*.

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*.

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.

OpenCores. (2012). *OpenRISC 1000 Architecture Manual, Architecture Version 1.0*.

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.

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

- 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.
- 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.
- Ungar, D., Blau, R., Foley, P., Samples, D., & Patterson, D. (1984). Architecture of SOAR: Smalltalk on a RISC. *ISCA*, 188–197.
- 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.



RISC-V[®]