RISC-V指令集手册

第I卷：无特权的ISA

文档版本20191214 *草案*

编辑：安德鲁·沃特曼[1]，Krste Asanović[1,2]

[1]个 SiFive Inc.，

[2] 加州大学伯克利分校EECS系CS系

andrew@sifive.com ，krste@berkeley.edu

2020年9月2日

该规范所有版本的贡献者均按字母顺序排列（请联系编辑以提出更正）：Arvind，Krste Asanovic，Rimas Avi˘žienis，Jacob Bachmeyer，Christopher F. Batten，Allen J.Baum，Alex Bradbury，Scott Beamer，Preston Briggs ，克里斯托弗·西里奥（Christopher Celio），张传华（Chuanhua Chang），大卫·奇斯纳尔（David Chisnall），保罗·克莱顿（Paul Clayton），帕尔默·达伯特（Ken Dockser），罗杰·埃斯帕萨（Roger Espasa），摇晃的Flur，斯蒂芬·弗洛登伯格（Stefan Freudenberger），马克·高迪耶（Marc Gauthier），安迪·格鲁（Andy Glew），扬·格雷（Jack Gray），迈克尔·汉堡，约翰·豪瑟（John Hauser），大卫·霍纳（David Horner），布鲁斯·霍特（Bruce Hoult），比尔霍夫曼，亚历山大·乔安努，奥洛夫·约翰逊，本·凯勒，大卫·克鲁米耶，李云s，保罗·洛文斯泰因，丹尼尔·卢斯蒂格，亚丁·曼纳尔卡，卢克·马兰吉特，玛格丽特·马托诺西，约瑟夫·迈尔斯，维贾扬和纳加拉扬，里希尤尔·尼克尔，乔纳斯·奥伯豪瑟，史蒂芬·奥里尔Ou，John Ousterhout，David Patterson，Christopher Pulte，Jose Renau，Josh Scheid，Colin Schmidt，Peter Sewell，Susmit Sarkar，Michael Taylor，Wesley Terpstra，Matt Thomas，Tommy Thorn，Caroline Trippel，Ray VanDeWalker，Muralidaran Vijayaraghavan，Megan Wachs，AndrewWaterman，Robert Watson，Derek Williams，AndrewWright，Reinoud Zandijk和Sizhuo Zhang。

请引用为：“ RISC-V指令集手册，第I卷：用户级ISA，文档版本20191214 *草案*”，编辑Andrew Waterman和Krste Asanovi´c，RISC-V基金会，2019年12月。

前言

本文档介绍了RISC-V非特权体系结构。

标记为“已批准”的ISA模块目前已被批准。标有模块 *冰冻的*
在批准之前，预计不会有太大变化。标有模块
草案 有望在批准之前发生变化。

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

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

Preface to Document Version 20191213-Base-Ratified

This document describes the RISC-V unprivileged architecture.

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

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

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

The changes in this version of the document include:

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

Preface to Document Version 20190608-Base-Ratified

This document describes the RISC-V unprivileged architecture.

The RVWMOmemory model has been ratified at this time. The ISAmodules marked Ratified, have been ratified at this time. The modules marked *Frozen* are not expected to change significantly

before being put up for ratification. The modules marked *Draft* are expected to change before ratification.

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

Base	Version	Status
RVWMO	2.0	Ratified
RV32I	2.1	Ratified
RV64I	2.1	Ratified
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
Zifencei	2.0	Ratified
Zicsr	2.0	Ratified
M	2.0	Ratified
A	*2.0*	Frozen
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
N	*1.1*	*Draft*
Zam	*0.1*	*Draft*

The changes in this version of the document include:

- Moved description to Ratified for the ISA modules ratified by the board in early 2019.
- Removed the A extension from ratification.
- Changed document version scheme to avoid confusion with versions of the ISA modules.
- Incremented the version numbers of the base integer ISA to 2.1, reflecting the presence of the ratified RVWMO memory model and exclusion of FENCE.I, counters, and CSR instructions that were in previous base ISA.

- Incremented the version numbers of the F and D extensions to 2.2, reflecting that version 2.1 changed the canonical NaN, and version 2.2 defined the NaN-boxing scheme and changed the definition of the FMIN and FMAX instructions.

- Changed name of document to refer to “unprivileged” instructions as part of move to separate ISA specifications from platform profile mandates.
- Added clearer and more precise definitions of execution environments, harts, traps, and memory accesses.

- Defined instruction-set categories: *standard, reserved, custom, non-standard,* and *nonconforming.*

- 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:

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.

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

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 [2].
- 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 [26], 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 [25] 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.*

Contents

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 generalpurpose software development.

- Support for the revised 2008 IEEE-754 floating-point standard [8].
- 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 [16], RISC-II [9], SOAR [22], and SPUR [12] 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-Vcompatible 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 [14 , 15] 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 Chapters 2 and 5 , 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 4 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers, and which has half the number of integer registers. Chapter 6 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 ADDD for 64-bit adds in RV64I, instead of the existing design which uses the same opcode ADD for 32bit 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 the Foundation; any extensions not defined by the Foundation 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 25 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 26 .

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

1.4 Memory

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

The execution environment determines the mapping of hardware resources into a hart’s address space. Different address ranges of a hart’s address space may (1) 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 14 and in appendices. Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in Chapter 23 . 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 16 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.

Figure 1.1 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.

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 Figure 1.1 . 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.

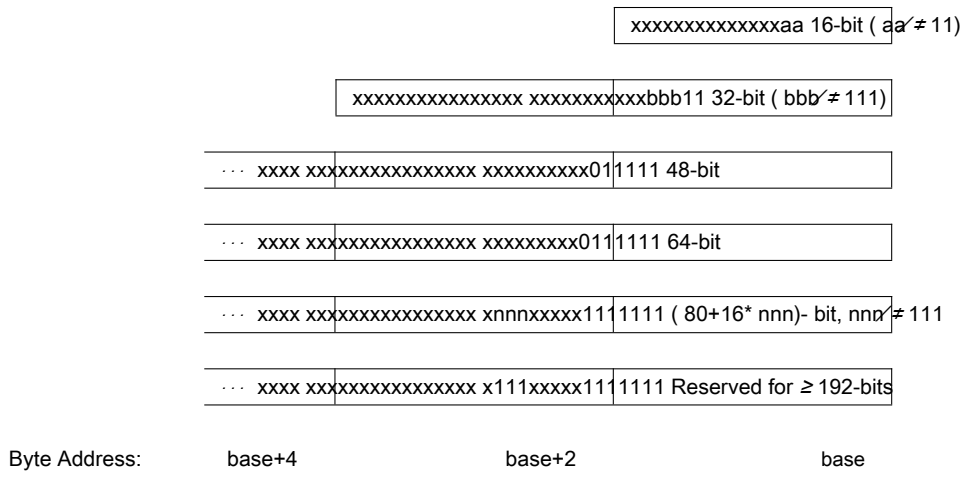


Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

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 in Chapter 25 , 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 littleendian 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 endiannessagnostic 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 supervisormode handler running on the same hart. Similarly, in the same environment, when a hart is interrupted, an interrupt handler will be run in supervisor mode on the hart.

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

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

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

Table 1.1 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 1.1: Characteristics of traps. Notes: 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 version 2.0 of 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 [1].

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

2.1 Programmers' Model for Base Integer ISA

Figure 2.1 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.

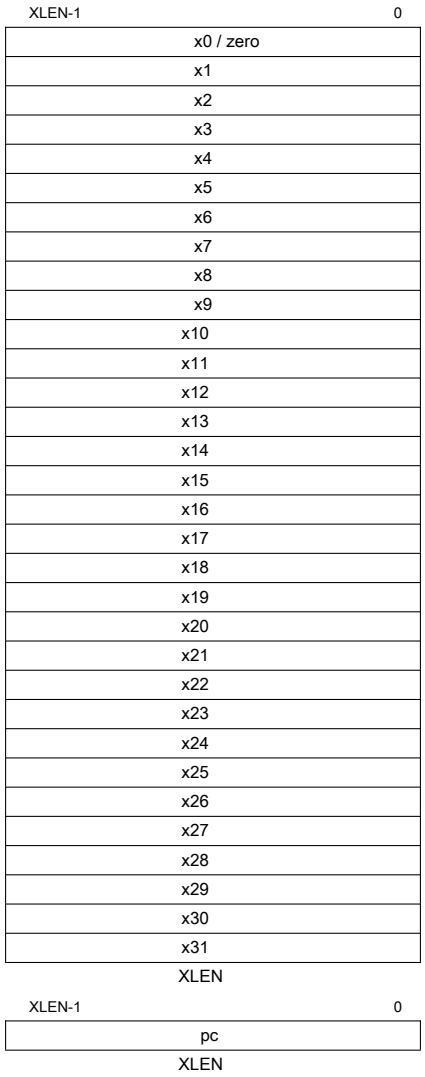


Figure 2.1: RISC-V base unprivileged integer register state.

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 the base ISA. 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 [21]. 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 4).

2.2 Base Instruction Formats

In the base RV32I ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2 . All are a fixed 32 bits in length and 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 four-byte 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 9), 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.

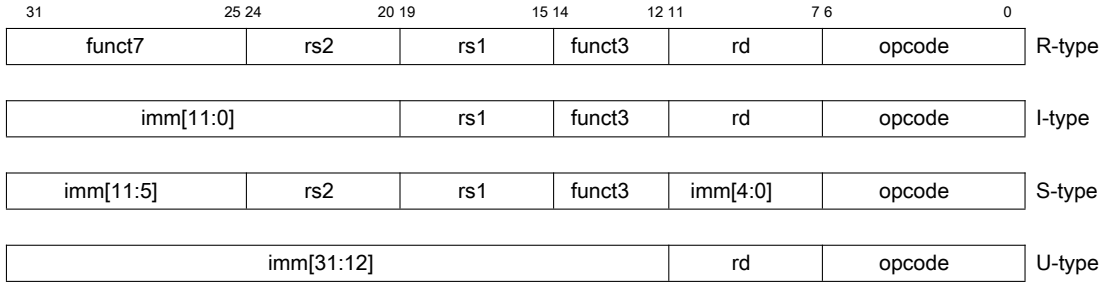


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[*x])* in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

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 [12]).

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 Figure 2.3 .

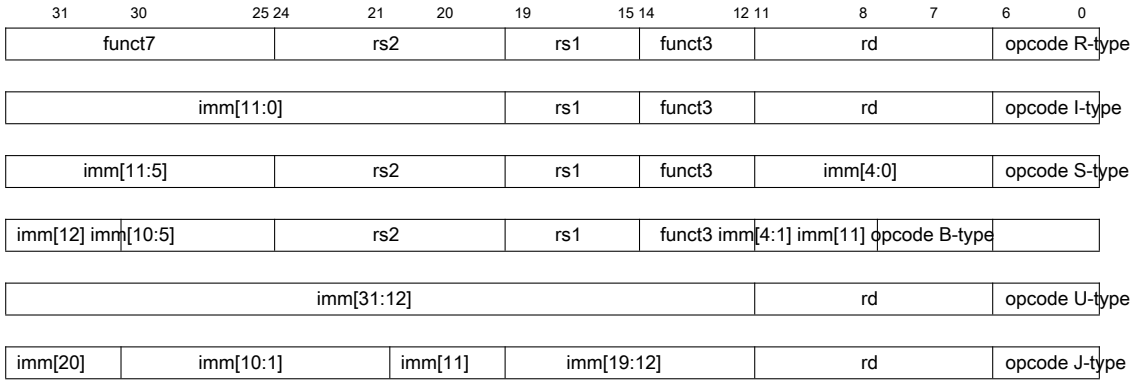


Figure 2.3: RISC-V base instruction formats showing immediate variants.

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.

Figure 2.4 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.

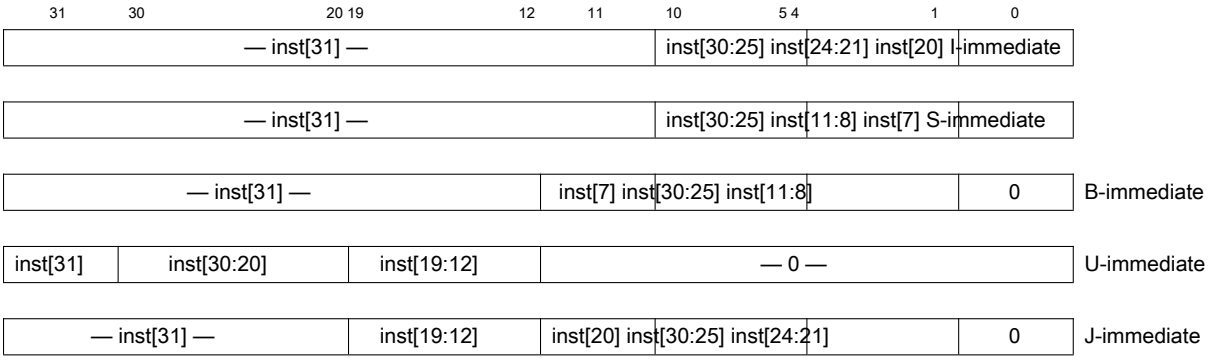


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

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.

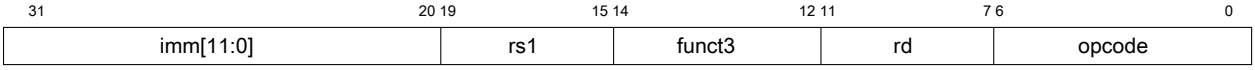
For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: addi t0, t1, +imm; blt t0, t1, overflow. *This covers the common case of addition with an immediate operand.*

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.

Integer Register-Immediate Instructions

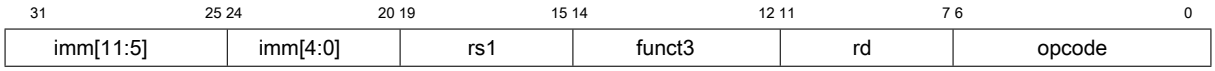


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

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.

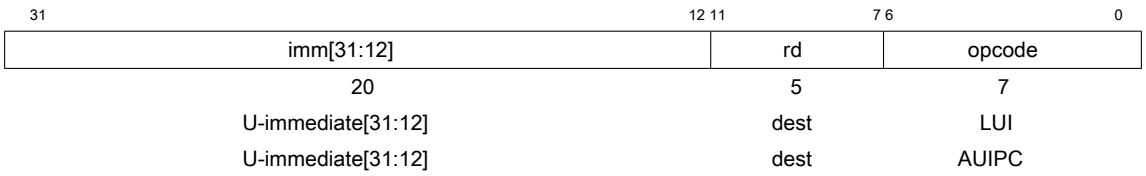
SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the signextended immediate when both are treated as signed numbers, else 0 is written to *rd.* SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 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).*



31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

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).



31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

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.*

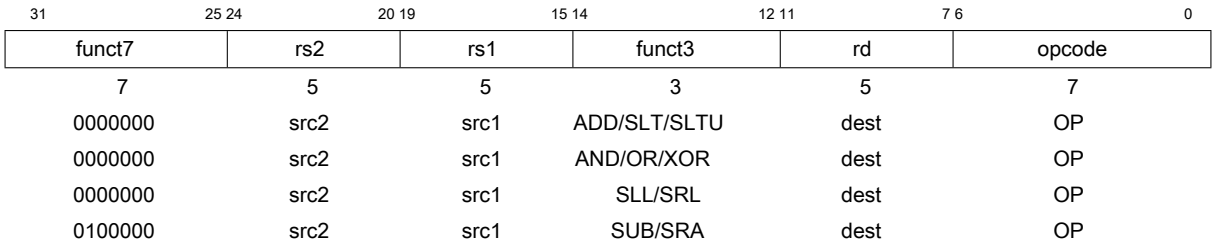
The assembly syntax for lui *and* auipc *does not represent the lower 12 bits of the U-immediate, which are always zero.*

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.

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.



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

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

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

NOP Instruction

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

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 (Section 2.9).

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.

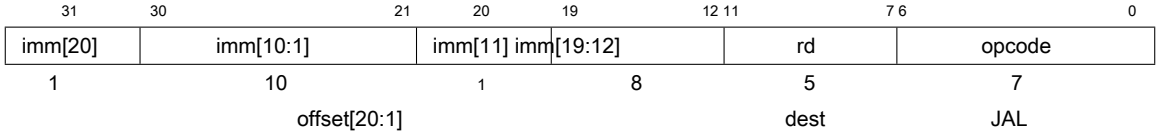
Unconditional Jumps

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ± 1MiB range.

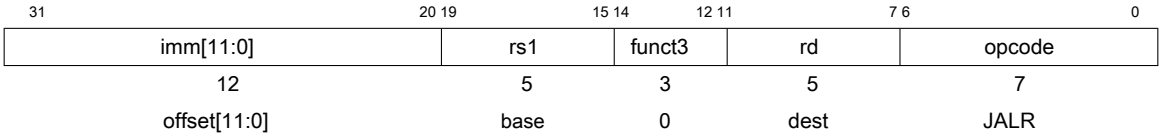
JAL stores the address of the instruction following the jump (pc+ 4) into register *rd.* The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

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

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



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



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

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

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

When used with a base rs1= x0, *JALR can be used to implement a single instruction subroutine call to the lowest 2KiB or highest 2KiB address region from anywhere in the address space, which could be used to implement fast calls to a small runtime library. Alternatively, an ABI could dedicate a general-purpose register to point to a library elsewhere in the address space.*

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

Instruction-address-misaligned exceptions are not possible on machines that support 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 the Table 2.1 .

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

Table 2.1: Return-address stack instruction. prediction hints encoded in the register operands of a JALR

Some other ISAs added explicit hint bits to their indirect-jump instructions to guide returnaddress 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)

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 ± 4KiB.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

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 alwaystrue 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 [7 , 11 , 10] and have been implemented in commercial processors [18]. 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 [18].

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

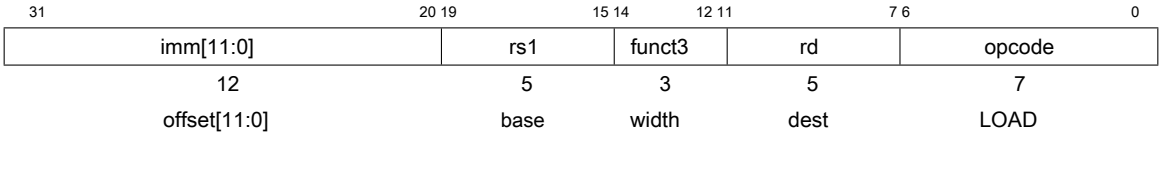
RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers. RV32I provides a 32-bit address space that is byte-addressed. The EEI will define what portions of the address space are legal to access with which instructions (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.

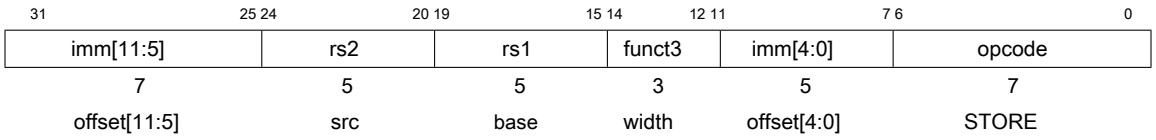
The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.

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 frommemory into *rd*. LH loads a 16-bit value frommemory, 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).

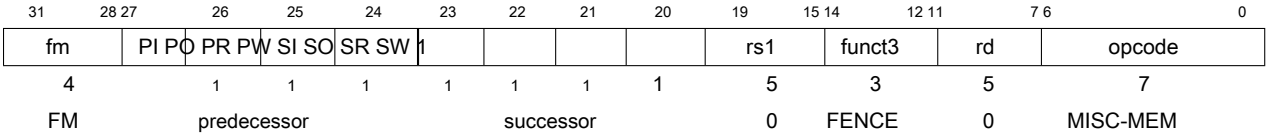
Misaligned accesses are occasionally required when porting legacy code, and help performance on applications when using any form of packed-SIMD extension or handling externally packed data structures. Our rationale for allowing EEIs to choose to support misaligned accesses via the regular load and store instructions is to simplify the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISA 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 RISCV 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 14 provides a precise description of the RISC-V memory consistency 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, memorymapped 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.

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

Table 2.2: Fence mode encoding.

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 optional 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.

The FENCE.TSO encoding was added as an optional extension to the original base FENCE instruction encoding. The base definition requires that implementations ignore any set bits and treat the FENCE as global, and so this is a backwards-compatible extension.

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 2.2 are also reserved for future use. Base implementations shall treat all such reserved

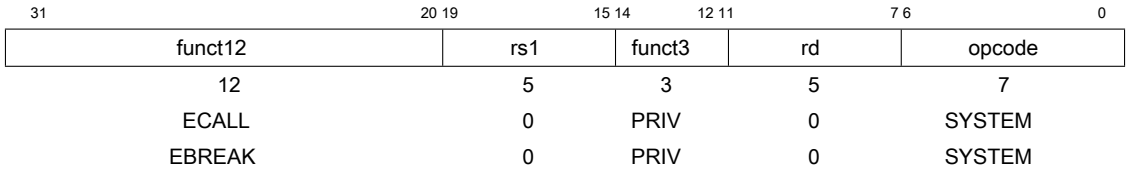
configurations as normal fences with *fm=* 0000, and standard software shall use only non-reserved configurations.

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 9 , 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.



31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

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

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

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

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

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

Another use of EBREAK is to support “semihosting”, where the execution environment includes a debugger that can provide services over an alternate system call interface built around the EBREAK instruction. Because the RISC-V base ISA does 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 ebreak        # Entry NOP
                                # 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 16 .

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. HINTs are encoded as integer computational instructions with *rd=* x0. Hence, 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.

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. 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.*

Table 2.3 lists all RV32I 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.

No standard hints are presently defined. 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.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*= x0	2_{20}	*Reserved for future standard use*
AUIPC	*rd*= x0	2_{20}	
ADDI	*rd*= x0, and either *rs1*≠ x0 or *imm*≠ 0	$2_{17}-1$	
ANDI	*rd*= x0	2_{17}	
ORI	*rd*= x0	2_{17}	
XORI	*rd*= x0	2_{17}	
ADD	*rd*= x0	2_{10}	
SUB	*rd*= x0	2_{10}	
AND	*rd*= x0	2_{10}	
OR	*rd*= x0	2_{10}	
XOR	*rd*= x0	2_{10}	
SLL	*rd*= x0	2_{10}	
SRL	*rd*= x0	2_{10}	
SRA	*rd*= x0	2_{10}	
FENCE	*pred*= 0 or *succ*= 0	2_5-1	
SLTI	*rd*= x0	2_{17}	*Designated for custom use*
SLTIU	*rd*= x0	2_{17}	
SLLI	*rd*= x0	2_{10}	
SRLI	*rd*= x0	2_{10}	
SRAI	*rd*= x0	2_{10}	
SLT	*rd*= x0	2_{10}	
SLTU	*rd*= x0	2_{10}	

Table 2.3: RV32I HINT instructions.

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 MAJC [20]). 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.

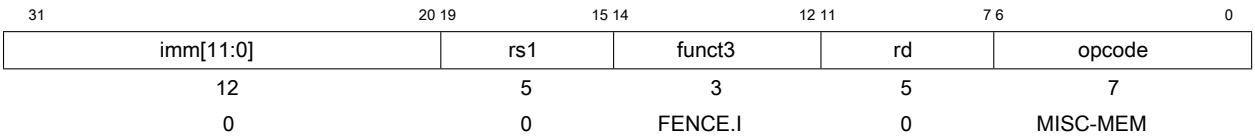
The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.

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.*



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	FENCE.I	0	MISC-MEM

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.

Chapter 4

RV32E Base Integer Instruction Set, Version 1.9

This chapter describes a draft proposal for the RV32E base integer instruction set, which is a reduced version of RV32I designed for embedded systems. The only change is to reduce the number of integer registers to 16. This chapter only outlines the differences between RV32E and RV32I, and so should be read after Chapter 2 .

RV32E was designed to provide an even smaller base core for embedded microcontrollers. Although we had mentioned this possibility in version 2.0 of this document, we initially resisted defining this subset. However, given the demand for the smallest possible 32-bit microcontroller, and in the interests of preempting fragmentation in this space, we have now defined RV32E as a fourth standard base ISA in addition to RV32I, RV64I, and RV128I. There is also interest in defining an RV64E to reduce context state for highly threaded 64-bit processors.

4.1 RV32E Programmers' Model

RV32E reduces 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 designs, 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.

This change requires a different calling convention and ABI. In particular, RV32E is only used with a soft-float calling convention. A new embedded ABI is under consideration that would work across RV32E and RV32I.

4.2 RV32E Instruction Set

RV32E uses the same instruction-set encoding as RV32I, except that only registers x0 – x15 are provided. Any future standard extensions will not make use of the instruction bits freed up by the reduced register-specifier fields and so these are designated for custom extensions.

RV32E can be combined with all current standard extensions. Defining the F, D, and Q extensions as having a 16-entry floating point register file when combined with RV32E was considered but decided against. To support systems with reduced floating-point register state, we intend to define a "Zfinx" extension that makes floating-point computations use the integer registers, removing the floating-point loads, stores, and moves between floating point and integer registers.

Chapter 5

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.

5.1 Register State

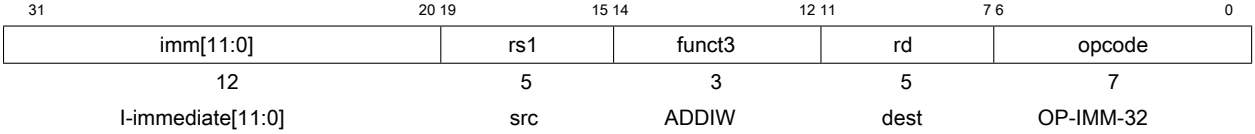
RV64I widens the integer registers and supported user address space to 64 bits (XLEN=64 in Figure 2.1).

5.2 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.

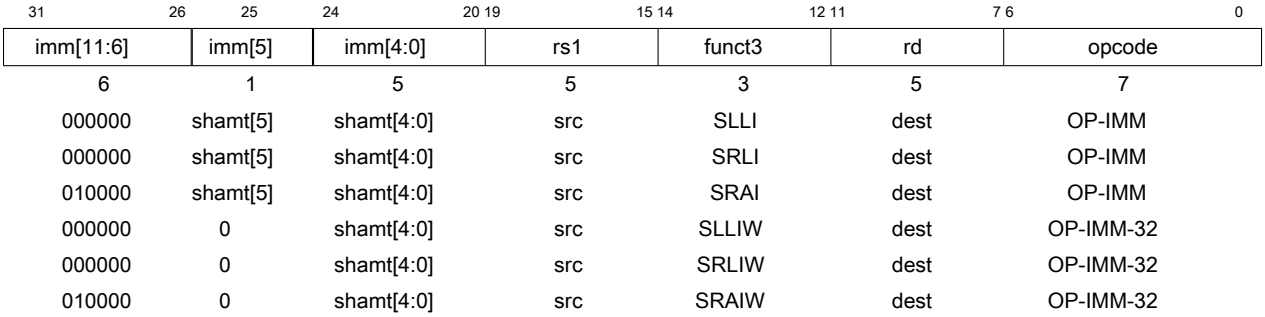
The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion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

Integer Register-Immediate Instructions



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDIW	dest	OP-IMM-32

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).

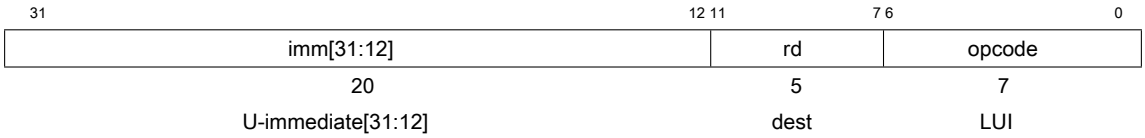


31 26	25	24 20	19 15	14 12	11 7	6 0
imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

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

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

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



31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

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 [$-2_{31} - 2_{11}, 2_{31} - 2_{11} - 1$].

Integer Register-Register Operations

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SRA	dest	OP
0000000	src2	src1	ADDW	dest	OP-32
0000000	src2	src1	SLLW/SRLW	dest	OP-32
0100000	src2	src1	SUBW/SRAW	dest	OP-32

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].*

5.3 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.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

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.

5.4 HINT Instructions

All instructions that are microarchitectural HINTs in RV32I (see Section 2.9) are also HINTs in RV64I. The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.

Table 5.1 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.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*= x0	2_{20}	*Reserved for future standard use*
AUIPC	*rd*= x0	2_{20}	
ADDI	*rd*= x0, and either *rs1*≠ x0 or *imm*≠ 0	$2_{17}-1$	
ANDI	*rd*= x0	2_{17}	
ORI	*rd*= x0	2_{17}	
XORI	*rd*= x0	2_{17}	
ADDIW	*rd*= x0	2_{17}	
ADD	*rd*= x0	2_{10}	
SUB	*rd*= x0	2_{10}	
AND	*rd*= x0	2_{10}	
OR	*rd*= x0	2_{10}	
XOR	*rd*= x0	2_{10}	
SLL	*rd*= x0	2_{10}	
SRL	*rd*= x0	2_{10}	
SRA	*rd*= x0	2_{10}	
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	*pred*= 0 or *succ*= 0	2_5-1	
SLTI	*rd*= x0	2_{17}	*Designated for custom use*
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}	

Table 5.1: RV64I HINT instructions.

Chapter 6

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 128bit 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.

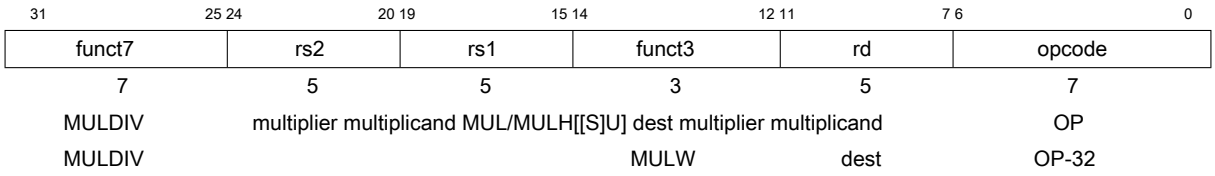
Chapter 7

“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.

7.1 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 signed *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

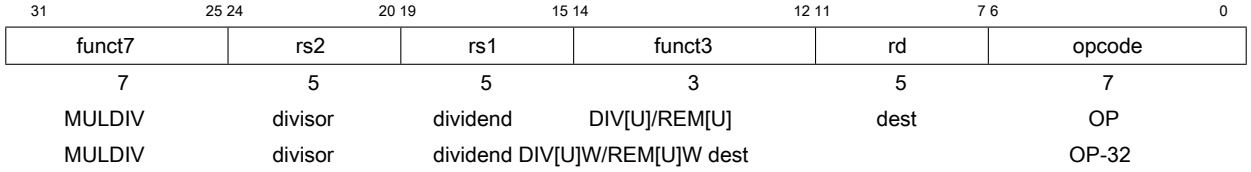
rdl, rs1, rs2 (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2).* Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

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].

7.2 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 the result equals the sign of the dividend.

For both signed and unsigned division, it holds that dividend = divisor × quotient + remainder.

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

DIVW and DIVUW are RV64 instructions that divide the lower 32 bits of *rs1* by the lower 32 bits of *rs2,* treating them as signed and unsigned integers 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 7.1 . 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.

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

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

Table 7.1: 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.

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.

Chapter 8

“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/storeconditional 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 [6].

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.

8.1 Specifying Ordering of Atomic Instructions

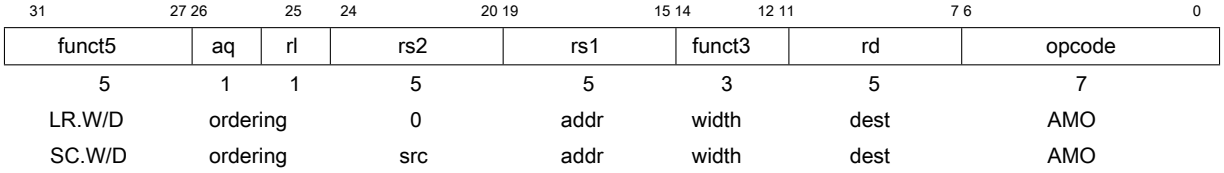
The base RISC-V ISA has a relaxed memory model, with the FENCE instruction used to impose additional ordering constraints. The address space is divided by the execution environment into memory and I/O domains, and the FENCE instruction provides options to order accesses to one or both of these two address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

To provide more efficient support for release consistency [6], 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.

8.2 Load-Reserved/Store-Conditional Instructions



31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

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.* 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 accesses 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. Our current thoughts are to include a small limited-capacity transactional memory

buffer along the lines of the original transactional memory proposals as an optional standard extension "T".

The failure code with value 1 is reserved to encode an unspecified failure. Other failure codes are reserved at this time, and 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 64-bit words and four-byte aligned for 32-bit 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 14.1 .

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. For example, the Unix platform is expected to require of main memory that the reservation set be of fixed size, contiguous, naturally aligned, and no greater than the virtual memory page size.

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 14.1 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 setting the *rl*
bit on the SC instruction. Setting the *aq* bit on the LR instruction, and setting both the *aq* and the *rl* bit on the SC instruction makes the LR/SC sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

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

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

```
        # a0 holds address of memory location
        # a1 holds expected value
        # a2 holds desired value
        # a0 holds return value, 0 if successful, !0 otherwise
cas:
        lr.w t0, (a0)               # Load original value.
        bne t0, a1, fail sc.w t0, a2,  # Doesn't match, so fail.
        (a0) bnez t0, cas           # Try to update.
                                    # 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.
```

Figure 8.1: 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 Figure 8.1 . If inlined, compare-and-swap functionality need only take four instructions.

8.3 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. Floatingpoint 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 loadreserved 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.

8.4 Atomic Memory Operations

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

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO instructions atomically load a data value from the address in *rs1,* place the value into register *rd,*
apply a binary operator to the loaded value and the original value in *rs2,* then store the result back to the address in *rs1*. AMOs can either operate on 64-bit (RV64 only) or 32-bit words in memory.

对于RV64，32位AMO始终对位于其中的值进行符号扩展 *rd* 并忽略原始值的高32位 *rs2*。

对于AMO，A扩展名要求该地址位于 *rs1* 自然地与操作数的大小对齐（即，对于64位字而言是8字节对齐而对于32位字而言是4字节）。如果地址不是自然对齐的，则将生成地址未对齐的异常或访问错误异常。如果不应该模拟未对齐的访问，则可以为内存访问生成访问错误异常，否则该内存访问将能够完成（除非未对齐）。本章中介绍的“ Zam”扩展名 22 ，放宽了这一要求，并指定了未对齐的AMO的语义。

支持的操作包括交换，整数加，按位与，按位或，按位XOR以及有符号和无符号整数的最大值和最小值。在没有排序约束的情况下，这些AMO可用于实现并行约简操作，通常，通过写入以下内容来丢弃返回值： x0。

我们提供了获取和运行样式的原子基元，因为它们可以比LR / SC或CAS更好地扩展到高度并行的系统。一个简单的微体系结构可以使用LR / SC原语来实现AMO，前提是该实现可以保证AMO最终完成。更复杂的实现也可能在内存控制器上实现AMO，并且可以在目标为 x0。

选择该组AMO是为了有效支持C11 / C ++ 11原子存储操作，并支持并行减少内存。AMO的另一用途是为I / O空间中的内存映射设备寄存器提供原子更新（例如，设置，清除或切换位）。

为了帮助实现多处理器同步，AMO可以选择提供发行版一致性语义。如果 *水* 位置1，则在AMO之前不会观察到此RISC-V暂存器中的后续存储操作。相反，如果 *rl* 该位置1时，则其他RISC-V挂接在该RISC-V挂接中的AMO之前的内存访问之前将不会观察到AMO。同时设置 *水* 和 *rl* AMO上的位使序列顺序一致，这意味着不能用同一存储单元中的较早或较晚的存储操作对其进行重新排序。

AMO旨在有效地实现C11和C ++ 11内存模型。尽管FENCE R，RW指令足以实现 获得 *操作和FENCE RW，W足以实施* 释放， *与具有相应的AMO相比，这都意味着额外的不必要的订购* 水 *要么* rl *设置。*

由测试和测试设置自旋锁保护的关键部分的示例代码序列如图2所示。 8.2 。请注意第一个AMO被标记 *水* 在关键部分之前命令获取锁，并标记第二个AMO *rl* 在放弃锁之前订购关键部分。

我们建议将以上显示的AMO Swap习惯用法用于锁获取和释放，以简化推测性锁省略的实现[17]。

“ A”扩展名中的指令还可用于提供顺序一致的加载和存储。可以将顺序一致的负载实现为LR *水* 和 *rl* 组。可以将顺序一致的存储实现为AMOSWAP，该存储将旧值写入x0并同时具有 *水* 和 *rl* 组。

```
        里                  t0，1               #初始化交换值。
再次：
        w                   t1，（a0）          #检查是否持有锁。
        Bnez                t1，再次            #重试（如果保持）。
        amoswap.w.aq t1，t0，（a0）#尝试获取锁定。Bnez
                            t1，再次            #重试（如果保持）。
        #。。。
        #关键部分。
        #。。。
        amoswap.w.rl x0，x0，（a0）#通过存储0释放锁定。
```

图8.2：互斥的示例代码。 00 包含锁的地址。

第九章

“ Zicsr”，控制和状态寄存器（CSR）指令，版本2.0

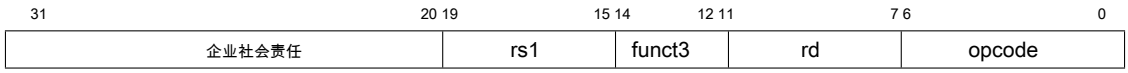
RISC-V定义了与每个hart关联的4096个控制和状态寄存器的单独地址空间。本章定义了在这些CSR上运行的完整CSR指令集。

虽然CSR主要由特权体系结构使用，但非特权代码中有多种用途，包括用于计数器和计时器以及浮点状态。

计数器和计时器不再被视为标准基本ISA的必需部分，因此访问它们所需的CSR指令已从基本ISA章节移至该单独的章节。

9.1企业社会责任说明

所有CSR指令以原子方式读取，修改，写入单个CSR，其CSR说明符编码为12位 *企业社会责任* 位31-20中保存的指令的字段。立即数格式使用5位零扩展立即数编码在 *rs1* 领域。



12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

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

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

Table 9.1: Conditions determining whether a CSR instruction reads or writes the specified CSR.

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, zeroextends 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. Other bits in the CSR are not explicitly written.

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zeroextends 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. Other bits in the CSR are not explicitly written.

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

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.

The CSRs defined so far in this volume do not have any architectural side effects on reads or writes. Custom extensions might add CSRs for which accesses have side effects.

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*.

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 explict 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 14), 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 MemoryOrdering 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 primarily to support ordering main memory and memory-mapped I/O accesses with respect to reads of the time *CSR. With the exception of the* time, cycle, *and* mcycle *CSRs, the CSRs defined thus far in Volumes I and II of this specification are not directly accessible to other harts or devices and cause no side effects visible to other harts or devices. Thus, accesses to CSRs other than the aforementioned three 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.

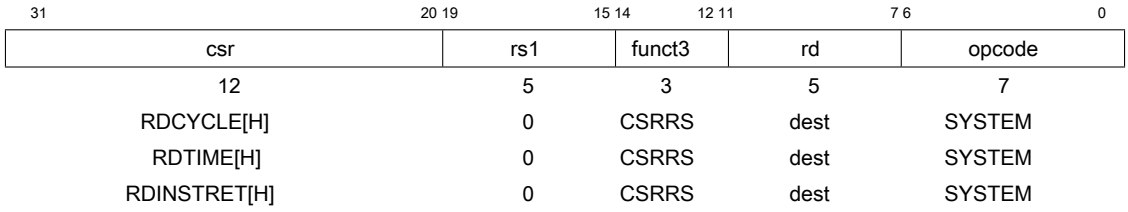
The rules for the reordering of CSR accesses in the global memory order should probably be moved to Chapter 14 concerning the RVWMO memory consistency model.

Chapter 10

Counters

RISC-V ISAs provide a set of up to 32 × 64-bit performance counters and timers that are accessible via unprivileged XLEN read-only CSR registers 0xC00 – 0xC1F (with the upper 32 bits accessed via CSR registers 0xC80 – 0xC9F on RV32). The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions (cycle count, real-time clock, and instructions-retired respectively), while the remaining counters, if implemented, provide programmable event counting.

10.1 Base Counters and Timers



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
RDCYCLE[H]	0	CSRRS	dest	SYSTEM
RDTIME[H]	0	CSRRS	dest	SYSTEM
RDINSTRET[H]	0	CSRRS	dest	SYSTEM

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12bit CSR address space and accessed in 32-bit pieces using CSRRS instructions. In RV64I, the CSR instructions can manipulate 64-bit CSRs. In particular, the RDCYCLE, RDTIME, and RDINSTRET pseudoinstructions read the full 64 bits of the cycle, time, and instret counters. Hence, the RDCYCLEH, RDTIMEH, and RDINSTRETH instructions are RV32I-only.

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

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 an RV32I-only instruction that 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 nonexistent 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 clockgated 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 an RV32I-only instruction that reads bits 63–32 of the same real-time counter. The underlying 64-bit counter should never overflow in practice. The execution environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant. The real-time clocks of all harts in a single user application should be synchronized to within one tick of the real-time clock. The environment should provide a means to determine the accuracy of the clock.

On some simple platforms, cycle count might represent a valid implementation of RDTIME, but in this case, platforms should implement the RDTIME instruction as an alias for RDCYCLE to make code more portable, rather than using RDCYCLE to measure wall-clock time.

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 an RV32I-only instruction that reads bits 63–32 of the same instruction counter. The underlying 64-bit counter should never overflow in practice.

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
```

Figure 10.1: Sample code for reading the 64-bit cycle counter in RV32.

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 should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

We required the counters be 64 bits wide, even on RV32, 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 described above shows how the full 64-bit width value can be safely read using the individual 32-bit instructions.

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.

10.2 Hardware Performance Counters

There is CSR space allocated for 29 additional unprivileged 64-bit hardware performance counters, hpmcounter3 – hpmcounter31. For RV32, the upper 32 bits of these performance counters is accessible via additional CSRs hpmcounter3h – hpmcounter31h. These counters count platformspecific events and are configured via additional privileged registers. The number and width of these additional counters, and the set of events they count is platform-specific.

The privileged architecture manual describes the privileged CSRs controlling access to these counters and to set the events to be counted.

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

Chapter 11

“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 [8]. The F extension depends on the “Zicsr” extension for control and status register access.

11.1 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 Figure 11.1 . 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.

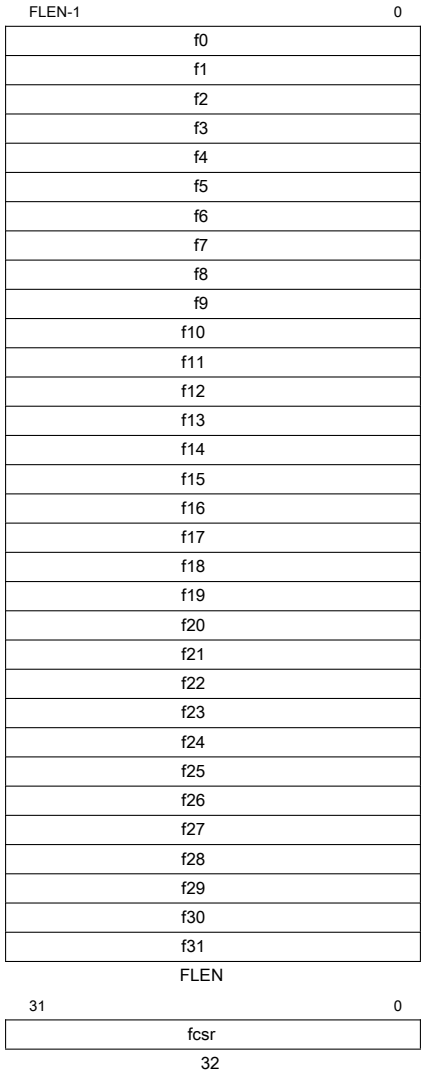


Figure 11.1: RISC-V standard F extension single-precision floating-point state.

11.2 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 Figure 11.2 .

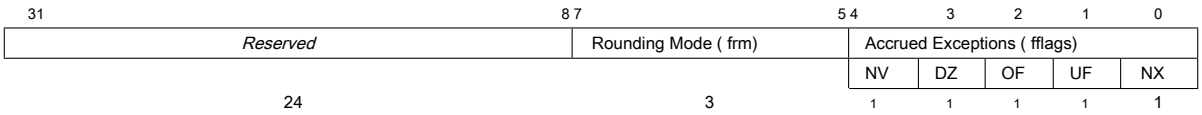


Figure 11.2: Floating-point control and status register.

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 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.

Bits 31–8 of the fcsr are reserved for other standard extensions, including the “L” standard extension for decimal floating-point. 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 11.1 . A value of 111 in the instruction’s *rm* field selects the dynamic rounding mode held in frm. If frm is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception. Some instructions, including widening conversions, have the *rm* field but are nevertheless unaffected by the rounding mode; software should set their *rm* field to RNE (000).

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.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even Round
001	RTZ	towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use. Invalid.*
110		*Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

Table 11.1: Rounding mode encoding.

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

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Table 11.2: Accrued exception flag encoding.

As allowed by the standard, we do not support traps on floating-point exceptions in the base ISA, 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.

11.3 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.

11.4 Subnormal Arithmetic

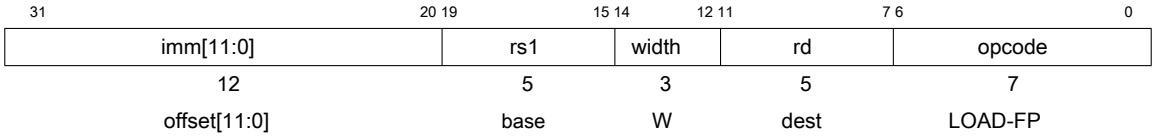
Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

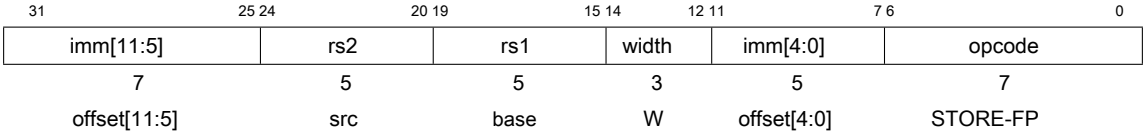
In the parlance of the IEEE standard, tininess is detected after rounding.

Detecting tininess after rounding results in fewer spurious underflow signals.

11.5 Single-Precision Load and Store Instructions

Floating-point loads and stores use the same base+offset addressing mode as the integer base ISA, 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 FSWdo not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

As described in Section 2.6 , the EEI defines whether misaligned floating-point loads and stores are handled invisibly or raise a contained or fatal trap.

11.6 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 floatingpoint 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 11.3 . It is set to *S* (00) for all instructions in the F extension.

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

Table 11.3: Format field encoding.

All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in Table 11.1 .

Floating-point minimum-number and maximum-number instructions FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. For the purposes of these instructions only, the value −0.0 is considered to be less than the value +0.0. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

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

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	S	src2	src1	RM	dest	OP-FP
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP
FSQRT	S	0	src	RM	dest	OP-FP
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP

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

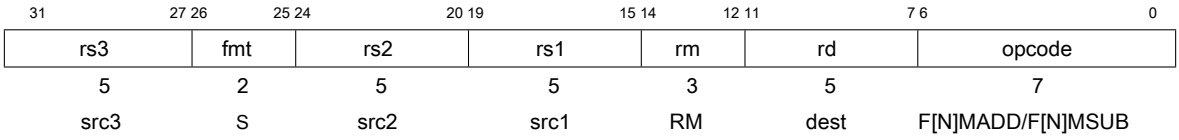
FMADD.S multiplies the values in *rs1* and *rs2,* adds the value in *rs3,* and writes the final result to *rd.* FMADD.S computes (*rs1 × rs2)+rs3.*

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

FNMSUB.S multiplies the values in *rs1* and *rs2,* negates the product, adds the value in *rs3,* and writes the final result to *rd.* FNMSUB.S computes -(*rs1 × rs2)+rs3.*

FNMADD.S multiplies the values in *rs1* and *rs2,* negates the product, subtracts the value in *rs3,* and writes the final result to *rd.* FNMADD.S computes -(*rs1* × *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 ∞× 0 + qNaN.

11.7 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 floatingpoint 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 11.4 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.

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

Table 11.4: Domains of float-to-integer conversions and behavior for invalid inputs.

All floating-point conversion instructions raise the Inexact exception if the result differs from its operand value, yet is representable in the destination format.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT. *int.fmt*	S	W[U]/L[U]	src	RM	dest	OP-FP
FCVT. *fmt.int*	S	W[U]/L[U]	src	RM	dest	OP-FP

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).*

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	S	src2	src1	J[N]/JX	dest	OP-FP

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 7542008 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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

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 base ISA 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.

11.8 Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers (*rs1 = rs2, rs1 < rs2, rs1 ≤ rs2)* writing 1 to the integer register *rd*
if the condition holds, and 0 otherwise.

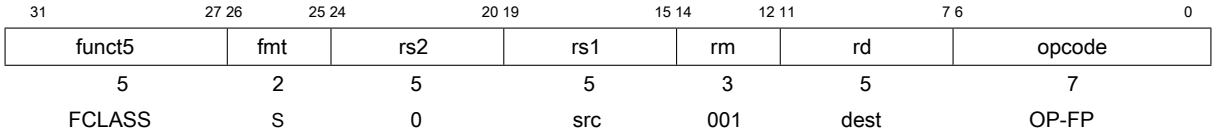
FLT.S and FLE.S perform what the IEEE 754-2008 standard refers to as *signaling* comparisons: that is, they set the invalid operation exception flag if either input is NaN. FEQ.S performs a *quiet*
comparison: it only sets the invalid operation exception flag if either input is a signaling NaN. For all three instructions, the result is 0 if either operand is NaN.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

The F extension provides a ≤ comparison, whereas the base ISA provides a ≥ branch comparison. Because ≤ can be synthesized from ≥ and vice-versa, there is no performance implication to this inconsistency, but it is nevertheless an unfortunate incongruity in the ISA.

11.9 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 11.5 . 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.



rd bit	Meaning
0	*rs1* is $-\infty$.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is +0.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

Table 11.5: Format of result of FCLASS instruction.

Chapter 12

“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.

12.1 D Register State

The D extension widens the 32 floating-point registers, f0 – f31, to 64 bits (FLEN=64 in Figure 11.1). The f registers can now hold either 32-bit or 64-bit floating-point values as described below in Section 12.2 .

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.

12.2 NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower n- bit types, $n <$ FLEN, are represented in the lower n bits of an FLEN-bit NaN value, in a process termed NaN-boxing. The upper bits of a valid NaN-boxed value must be all 1s. Valid NaN-boxed n- bit values therefore appear as negative quiet NaNs (qNaNs) when viewed as any wider m- bit value, $n < m \leq$ FLEN. Any operation that writes a narrower result to an f register must write all 1s to the uppermost FLEN $- n$ bits to yield a legal NaN-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 floatingpoint move instructions (FMV. *n.* X/FMV.X. *n).* A narrower *n*- bit transfer, *n* < FLEN, into the f
registers will create a valid NaN-boxed value. A narrower *n*- bit transfer out of the floating-point registers will transfer the lower *n* bits of the register ignoring the upper FLEN − *n* bits.

Apart from transfer operations described in the previous paragraph, all other floating-point operations on narrower *n*- bit operations, *n* < FLEN, check if the input operands are correctly NaN-boxed,
i.e., all upper FLEN − *n* bits are 1. If so, the *n* least-significant bits of the input are used as the input value, otherwise the input value is treated as an *n*- bit canonical NaN.

Earlier versions of this document did not define the behavior of feeding the results of narrower or wider operands into an operation, except to require that wider saves and restores would preserve the value of a narrower operand. The new definition removes this implementation-specific behavior, while still accommodating both non-recoded and recoded implementations of the floating-point unit. The new definition also helps catch software errors by propagating NaNs if values are used incorrectly.

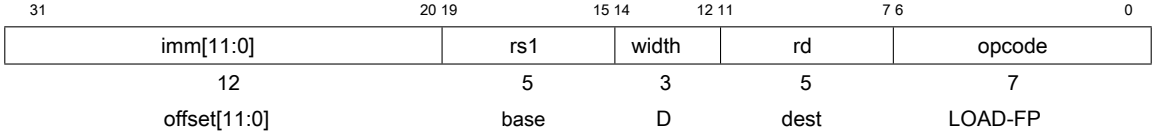
Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in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

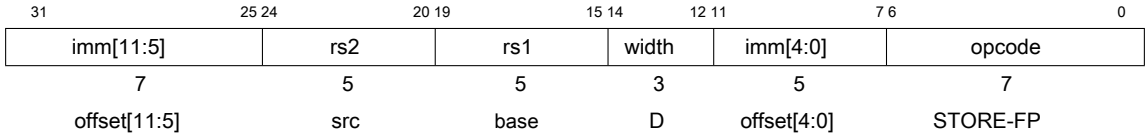
Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can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

12.3 Double-Precision Load and Store Instructions

The FLD instruction loads a double-precision floating-point value from memory into floating-point register *rd.* FSD stores a double-precision value from the floating-point registers to memory.

The double-precision value may be a NaN-boxed single-precision value.



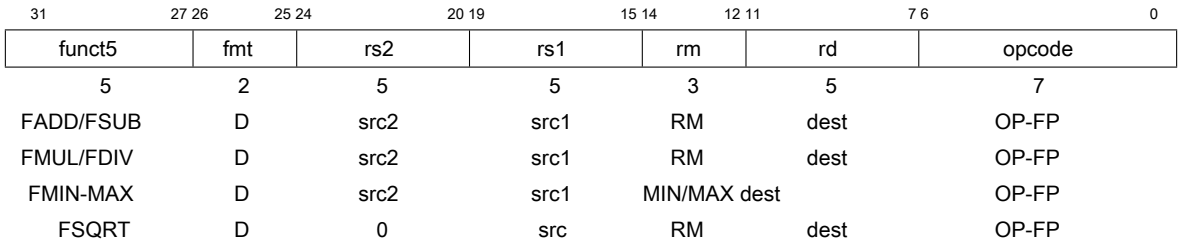


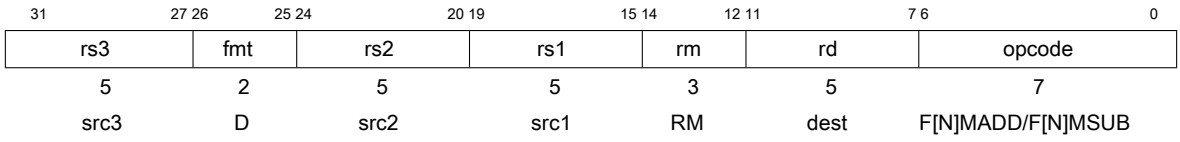
FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN ≥ 64.

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

12.4 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 doubleprecision results.





12.5 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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT. *int.* D	D	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.D. *int*	D	W[U]/L[U]	src	RM	dest	OP-FP

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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.D	S	D	src	RM	dest	OP-FP
FCVT.D.S	D	S	src	RM	dest	OP-FP

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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

For XLEN ≥ 64 only, instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.D moves the double-precision value in floating-point register *rs1* to a representation in IEEE 754-2008 standard encoding in integer register *rd.* FMV.D.X moves the double-precision value encoded in IEEE 754-2008 standard encoding from the integer register *rs1*

to the floating-point register *rd.*

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

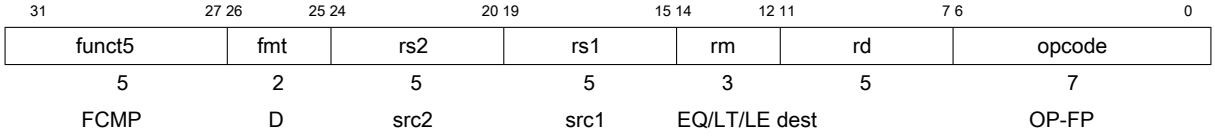
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.D	D	0	src	000	dest	OP-FP
FMV.D.X	D	0	src	000	dest	OP-FP

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 multiplyadd 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 addressspace and pointers can be used to avoid growth of static data and dynamic memory traffic.

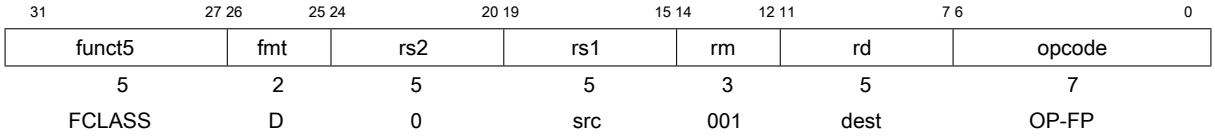
12.6 Double-Precision Floating-Point Compare Instructions

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



12.7 Double-Precision Floating-Point Classify Instruction

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



Chapter 13

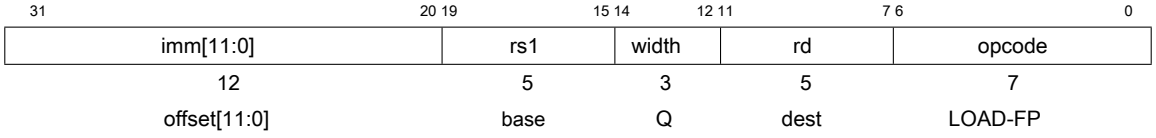
“Q” Standard Extension for Quad-Precision Floating-Point, Version 2.2

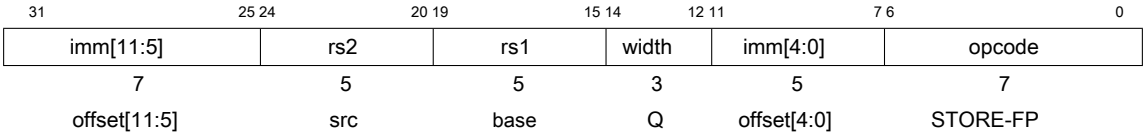
This chapter describes the Q standard extension for 128-bit quad-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The quad-precision binary floating-point instruction-set extension is named “Q”; it depends on the double-precision floatingpoint extension 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 12.2

is now extended recursively to allow a single-precision value to be NaN-boxed inside a doubleprecision value which is itself NaN-boxed inside a quad-precision value.

13.1 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.

13.2 Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 13.1 .

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

Table 13.1: Format field encoding.

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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FADD/FSUB	Q	src2	src1	RM	dest	OP-FP
FMUL/FDIV	Q	src2	src1	RM	dest	OP-FP
FMIN-MAX	Q	src2	src1	MIN/MAX	dest	OP-FP
FSQRT	Q	0	src	RM	dest	OP-FP

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	Q	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

13.3 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-doubleprecision conversion instructions. FCVT.W.Q or FCVT.L.Q converts a quad-precision floatingpoint 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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT. *int.* Q	Q	W[U]/L[U]	src	RM	dest	OP-FP
FCVT.Q. *int*	Q	W[U]/L[U]	src	RM	dest	OP-FP

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 quadprecision floating-point number to a double-precision floating-point number, or vice-versa, respectively.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCVT.S.Q	S	Q	src	RM	dest	OP-FP
FCVT.Q.S	Q	S	src	RM	dest	OP-FP
FCVT.D.Q	D	Q	src	RM	dest	OP-FP
FCVT.Q.D	Q	D	src	RM	dest	OP-FP

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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	Q	src2	src1	J[N]/JX	dest	OP-FP

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.

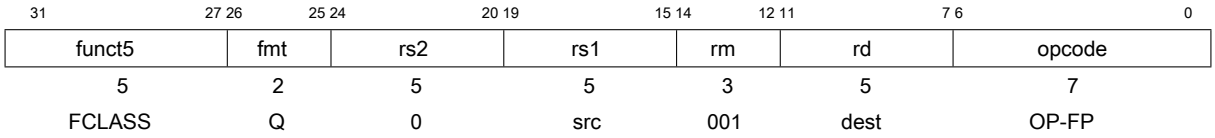
13.4 Quad-Precision Floating-Point Compare Instructions

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

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	Q	src2	src1	EQ/LT/LE	dest	OP-FP

13.5 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 14

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 loadreserved/store-conditional and atomic read-modify-write instructions.

The standard ISA extension for misaligned atomics “Zam” (Chapter 22) and the standard ISA extension for total store ordering “Ztso” (Chapter 23) augment 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, “T” transactional memory, 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.

14.1 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).

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 12.3 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 "Zam" extension, LRs, SCs, and AMOs may be decomposed subject to the constraints of the atomicity axiom for misaligned atomics, which is defined in Chapter 22 .

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 8.2 .

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 “releaseRCsc” 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 releaseRCpc 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 processorconsistent synchronization operations, and the term “RCsc” stands for release consistency with sequentially-consistent synchronization operations [6].

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 23). 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.

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 14.2 .

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 14.3 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 14.3
- *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 14.3
- *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 14.3

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.*

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*

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.

14.2 CSR Dependency Tracking Granularity

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

Table 14.1: Granularities at which syntactic dependencies are tracked through CSRs

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

14.3 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 14.1 .

The term “accumulating CSR” is used to describe a CSR that is both a source and a destination register, but which carries a dependency only from itself to itself.

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

Key:

A Address source register

D Data source register

↾ The instruction does not carry a dependency from any source register to any destination register

↧ The instruction carries dependencies from source register(s) to destination register(s) as specified

RV32I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs
LUI		*rd*	
AUIPC		*rd*	
JAL		*rd*	
JALR †	*rs1*	*rd*	
BEQ	*rs1, rs2*		
BNE	*rs1, rs2*		
BLT	*rs1, rs2*		
BGE	*rs1, rs2*		
BLTU	*rs1, rs2*		
BGEU	*rs1, rs2*		
LB †	*rs1* $_A$	*rd*	
LH †	*rs1* $_A$	*rd*	
LW †	*rs1* $_A$	*rd*	
LBU †	*rs1* $_A$	*rd*	
LHU †	*rs1* $_A$	*rd*	
SB	*rs1* $_{A,}$ *rs2* $_D$		
SH	*rs1* $_{A,}$ *rs2* $_D$		
SW	*rs1* $_{A,}$ *rs2* $_D$		
ADDI	*rs1*	*rd*	
SLTI	*rs1*	*rd*	
SLTIU	*rs1*	*rd*	
XORI	*rs1*	*rd*	
ORI	*rs1*	*rd*	
ANDI	*rs1*	*rd*	
SLLI	*rs1*	*rd*	
SRLI	*rs1*	*rd*	
SRAI	*rs1*	*rd*	
ADD	*rs1, rs2*	*rd*	
SUB	*rs1, rs2*	*rd*	
SLL	*rs1, rs2*	*rd*	
SLT	*rs1, rs2*	*rd*	
SLTU	*rs1, rs2*	*rd*	
XOR	*rs1, rs2*	*rd*	
SRL	*rs1, rs2*	*rd*	
SRA	*rs1, rs2*	*rd*	
OR	*rs1, rs2*	*rd*	
AND	*rs1, rs2*	*rd*	
FENCE			
FENCE.I			
ECALL			
EBREAK			

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
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*

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
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

‡ carries a dependency from *csr* to *rd*

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*	

RV32M Standard Extension

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

RV64M Standard Extension

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

RV32A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs
LR.W †	*rs1*$_A$	*rd*	
SC.W †	*rs1*$_A$, *rs2*$_D$	*rd*$_*$	
AMOSWAP.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOADD.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOXOR.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOAND.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOOR.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMIN.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMAX.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMINU.W †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMAXU.W †	*rs1*$_A$, *rs2*$_D$	*rd*	

$_*$ if successful

RV64A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs
LR.D †	*rs1*$_A$	*rd*	
SC.D †	*rs1*$_A$, *rs2*$_D$	*rd*$_*$	
AMOSWAP.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOADD.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOXOR.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOAND.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOOR.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMIN.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMAX.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMINU.D †	*rs1*$_A$, *rs2*$_D$	*rd*	
AMOMAXU.D †	*rs1*$_A$, *rs2*$_D$	*rd*	

$_*$ if successful

RV32F Standard Extension

	Source Registers	Destination Accumulating Registers	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 NV, NX	* if rm=111
FSQRT.S	*rs1,* frm *	*rd*		* 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*		

RV64F Standard Extension

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

RV32D Standard Extension

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

RV64D Standard Extension

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

Chapter 15

“L” Standard Extension for Decimal Floating-Point, Version 0.0

This chapter is a draft proposal that has not been ratified by the Foundation.

This chapter is a placeholder for the specification of a standard extension named “L” designed to support decimal floating-point arithmetic as defined in the IEEE 754-2008 standard.

15.1 Decimal Floating-Point Registers

Existing floating-point registers are used to hold 64-bit and 128-bit decimal floating-point values, and the existing floating-point load and store instructions are used to move values to and from memory.

Due to the large opcode space required by the fused multiply-add instructions, the decimal floatingpoint instruction extension will require five 25-bit major opcodes in a 30-bit encoding space.

Chapter 16

“C” Standard Extension for Compressed Instructions, Version 2.0

This chapter describes the current proposal for 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.

16.1 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 16.4 , a few opcodes are used for different purposes depending on base ISA width. 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 register widths adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISA register widths. 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, 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 [5], 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 [4] 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 [19], 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 [23].*

16.2 Compressed Instruction Formats

Table 16.1 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 16.2 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 base ISA, which only has 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 example, immediate bits 17—10 are always sourced from the same instruction bit positions. Five other immediate bits (5, 4, 3, 1, and 0) have just two source instruction bits, while four (9, 7, 6, and 2) have three sources and one (8) has four sources.

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.

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 16.1: Compressed 16-bit RVC instruction 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

Table 16.2: Registers specified by the three-bit *rs1′*, *rs2′*, and *rd′* fields of the CIW, CL, CS, CA, and CB formats.

16.3 Load and Store Instructions

To increase the reach of 16-bit instructions, data-transfer instructions use zero-extended immediates that are scaled by the size of the data in bytes: ×4 for words, ×8 for double words, and ×16 for quad words.

RVC provides two variants of loads and stores. One uses the ABI stack pointer, x2, as the base address and can target any data register. The other can reference one of 8 base address registers and one of 8 data registers.

Stack-Pointer-Based Loads and Stores

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

These instructions use the CI format.

C.LWSP loads a 32-bit value from memory into register *rd*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, x2. It expands to lw rd, offset(x2). C.LWSP is 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).

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

These instructions use the CSS format.

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

C.SDSP is an RV64C/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 loadmultiple 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 [24].

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.

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 floatingpoint 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′).

16.4 Control Transfer Instructions

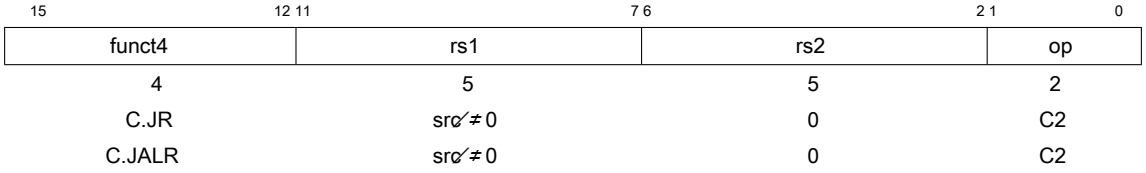
RVC provides unconditional jump instructions and conditional branch instructions. As with base RVI instructions, the offsets of all RVC control transfer instruction are in multiples of 2 bytes.

15 13	12 2	1 0
funct3	imm	op
3	11	2
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1

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 ±2KiB 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.



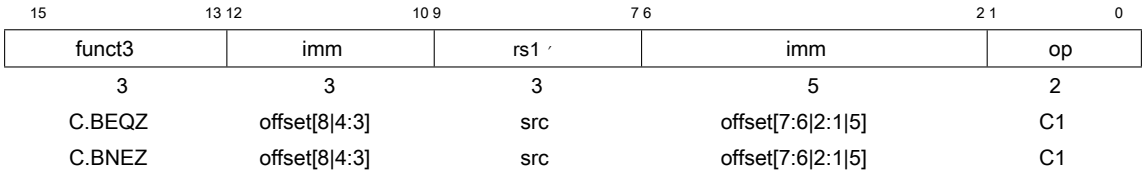
15 12	11 7	6 2	1 0
funct4	rs1	rs2	op
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

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*≠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*≠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.



15 13	12 10	9 7	6 2	1 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

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 ±256B range. C.BEQZ takes the branch if the value in register *rs1*′ is zero. It expands to beq rs1′,x0, offset.

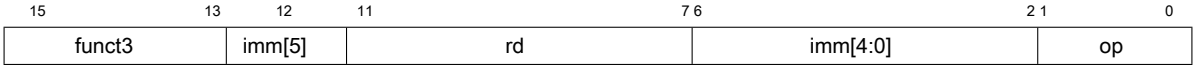
C.BNEZ is defined analogously, but it takes the branch if *rs1*′ contains a nonzero value. It expands
to bne rs1′,x0, offset.

16.5 Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

Integer Constant-Generation Instructions

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



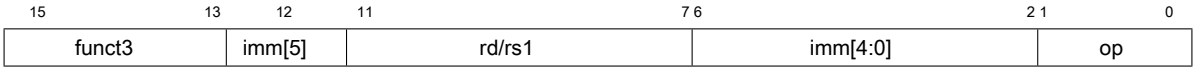
3	1	5	5	2
C.LI	imm[5]	dest ≠ 0	imm[4:0]	C1
C.LUI	nzimm[17]	dest ≠ {0, 2}	nzimm[16:12]	C1

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* ≠ 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, nzimm. C.LUI is only valid when *rd* ≠ {x0, x2}, and when the immediate is not equal to zero. The code points with *nzimm* = 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.

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.



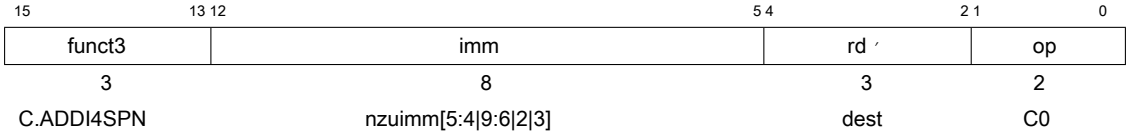
3	1	5	5	2
C.ADDI	nzimm[5]	dest ≠ 0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest ≠ 0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

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, nzimm. C.ADDI is only valid when *rd* ≠ x0
and *nzimm* ≠ 0. The code points with *rd* = x0 encode the C.NOP instruction; the remaining code points with *nzimm* = 0 encode HINTs.

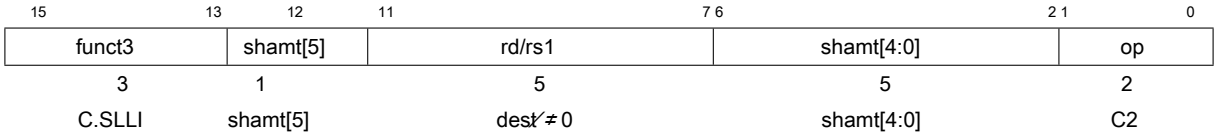
C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into addiw rd, rd, imm. The immediate can be zero for C.ADDIW, where this corresponds to sext.w rd. C.ADDIW is only valid when *rd* ≠ x0; the 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. 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. 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, except for RV128C with shamt=0, which expands to slli rd, rd, 64.

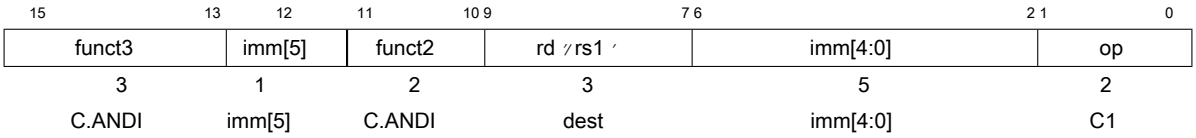
For RV32C, *shamt[5]* must be zero; the code points with *shamt[5]*= 1 are designated for custom extensions. For RV32C and RV64C, the shift amount must be non-zero; the code points with
shamt= 0 are HINTs. For all base ISAs, the code points with *rd*= x0 are HINTs, except those with
shamt[5]= 1 in RV32C.

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.

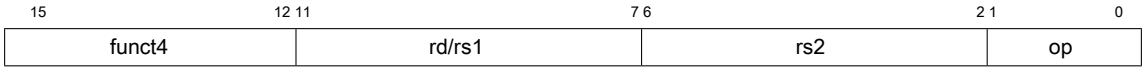
C.SRAI is defined analogously to C.SRLI, but instead performs an arithmetic right shift. C.SRAI expands to srai rd′, rd′, shamt.

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.

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*≠x0; the code points with *rs2*=x0 correspond to the C.JR instruction. The code points with *rs2*≠x0 and *rd*=x0 are HINTs.

C.MV expands to a different instruction than the canonical MV pseudoinstruction, which instead uses ADDI. Implementations that handle MV specially, e.g. using register-renaming hardware, may find it more convenient to expand C.MV to MV instead of ADD, at slight additional hardware cost.

C.ADD adds the values in registers *rd* and *rs2* and writes the result to register *rd*. C.ADD expands into add rd, rd, rs2. C.ADD is only valid when *rs2*≠x0; the code points with *rs2*=x0 correspond to the C.JALR and C.EBREAK instructions. The code points with *rs2*≠x0 and *rd*=x0 are HINTs.

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.

Defined Illegal Instruction

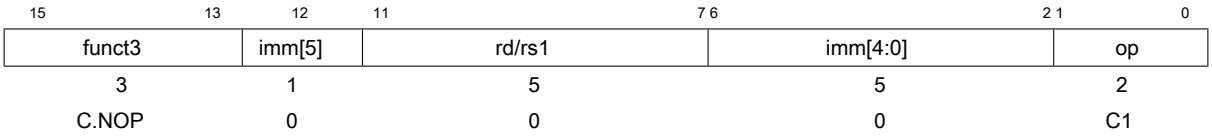
15 13	12	11 7	6 2	1 0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

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.

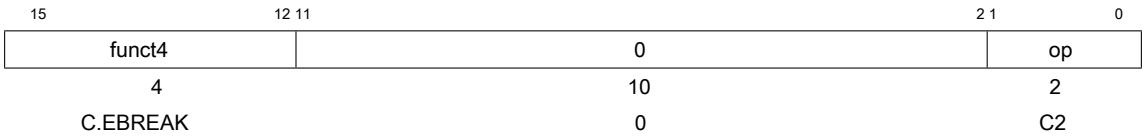
NOP Instruction



15–13	12	11–7	6–2	1–0
funct3	imm[5]	rd/rs1	imm[4:0]	op
3	1	5	5	2
C.NOP	0	0	0	C1

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.

Breakpoint Instruction



15–12	11–2	1–0
funct4	0	op
4	10	2
C.EBREAK	0	C2

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.

16.6 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 8.3 , 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.

16.7 HINT Instructions

A portion of the RVC encoding space is reserved for microarchitectural HINTs. Like the HINTs in the RV32I base ISA (see Section 2.9), 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, t0* might not encode the same HINT as ADD *x0, x0, t0*.

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 16.3 lists all RVC HINT code points. For RV32C, 78% 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.

Instruction	Constraints	Code Points	Purpose
C.NOP	*nzimm*′ ≠ 0	63	*Reserved for future standard use*
C.ADDI	*rd*′ ≠ x0, *nzimm* = 0	31	
C.LI	*rd* = x0	64	
C.LUI	*rd* = x0, *nzimm*′ ≠ 0	63	
C.MV	*rd* = x0, *rs2*′ ≠ x0	31	
C.ADD	*rd* = x0, *rs2*′ ≠ x0	31	
C.SLLI	*rd* = x0, *nzimm* = 0	31 (RV32) 63 (RV64/128)	*Designated for custom use*
C.SLLI64	*rd* = x0	1	
C.SLLI64	*rd*′ ≠ x0, RV32 and RV64 only RV32 and	31	
C.SRLI64	RV64 only	8	
C.SRAI64	RV32 and RV64 only	8	

Table 16.3: RVC HINT instructions.

16.8 RVC Instruction Set Listings

Table 16.4 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; *NSE* to indicate that the opcode is designated for custom extensions; or *HINT* to indicate that the opcode is reserved for microarchitectural hints (see Section 16.7).

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 SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LWSP LQSP		FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SWSP SQSP		FSWSP RV32 SDSP SDSP	RV64 RV128
11	> 16b								

Table 16.4: RVC opcode map

Tables 16.5 – 16.7 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	nzuimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN (*RES, nzuimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD (*RV32/64)*
001	uimm[5:4|8]	rs1′	uimm[7:6]	rd′	00	C.LQ (*RV128)*
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.FLW (*RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.LD (*RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD (*RV32/64)*
101	uimm[5:4|8]	rs1′	uimm[7:6]	rs2′	00	C.SQ (*RV128)*
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.FSW (*RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.SD (*RV64/128)*

Table 16.5: Instruction listing for RVC, Quadrant 0.

15 14 13	12	11 10 9	8 7	6 5	4 3 2	1 0	
000	nzimm[5]	0		nzimm[4:0]		01	C.NOP (*HINT, nzimm≠0)*
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI (*HINT, nzimm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]					01	C.JAL (*RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW (*RV64/128; RES, rd=0)*
010	imm[5]	rd≠0		imm[4:0]		01	C.LI (*HINT, rd=0)*
011	nzimm[9]			nzimm[4|6 /8:7|5]		01	C.ADDI16SP (*RES, nzimm=0)*
011	nzimm[17]	rd≠{2 0, 2}		nzimm[16:12]		01	C.LUI (*RES, nzimm=0; HINT, rd=0)*
100	nzuimm[5]	00	rs1 ′/rd ′	nzuimm[4:0]		01	C.SRLI (*RV32 NSE, nzuimm[5]=1)*
100	0	00	rs1 ′/rd ′	0		01	C.SRLI64 (*RV128; RV32/64 HINT)*
100	nzuimm[5]	01	rs1 ′/rd ′	nzuimm[4:0]		01	C.SRAI (*RV32 NSE, nzuimm[5]=1)*
100	0	01	rs1 ′/rd ′	0		01	C.SRAI64 (*RV128; RV32/64 HINT)*
100	imm[5]	10	rs1 ′/rd ′	imm[4:0]		01	C.ANDI
100	0	11	rs1 ′/rd ′	00	rs2 ′	01	C.SUB
100	0	11	rs1 ′/rd ′	01	rs2 ′	01	C.XOR
100	0	11	rs1 ′/rd ′	10	rs2 ′	01	C.OR
100	0	11	rs1 ′/rd ′	11	rs2 ′	01	C.AND
100	1	11	rs1 ′/rd ′	00	rs2 ′	01	C.SUBW (*RV64/128; RV32 RES)*
100	1	11	rs1 ′/rd ′	01	rs2 ′	01	C.ADDW (*RV64/128; RV32 RES)*
100	1	11	—	10	—	01	*Reserved*
100	1	11	—	11	—	01	*Reserved*
101	imm[11|4|9:8|10|6|7|3:1|5]					01	C.J
110	imm[8|4:3]		rs1 ′	imm[7:6|2:1|5]		01	C.BEQZ
111	imm[8|4:3]		rs1 ′	imm[7:6|2:1|5]		01	C.BNEZ

Table 16.6: Instruction listing for RVC, Quadrant 1.

15 14 13	12	11 10 9	8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0		nzuimm[4:0]	10	C.SLLI (*HINT, rd=0; RV32 NSE, nzuimm[5]=1)*
000	0	rs1/rd≠0		0	10	C.SLLI64 (*RV128; RV32/64 HINT; HINT, rd=0)*
001	uimm[5]	rd		uimm[4:3|8:6]	10	C.FLDSP (*RV32/64)*
001	uimm[5]	rd≠0		uimm[4|9:6]	10	C.LQSP (*RV128; RES, rd=0)*
010	uimm[5]	rd≠0		uimm[4:2|7:6]	10	C.LWSP (*RES, rd=0)*
011	uimm[5]	rd		uimm[4:2|7:6]	10	C.FLWSP (*RV32)*
011	uimm[5]	rd≠0		uimm[4:3|8:6]	10	C.LDSP (*RV64/128; RES, rd=0)*
100	0	rs1≠0		0	10	C.JR (*RES, rs1=0)*
100	0	rd≠0		rs2≠0	10	C.MV (*HINT, rd=0)*
100	1	0		0	10	C.EBREAK
100	1	rs1≠0		0	10	C.JALR
100	1	rs1/rd≠0		rs2≠0	10	C.ADD (*HINT, rd=0)*
101	uimm[5:3|8:6]			rs2	10	C.FSDSP (*RV32/64)*
101	uimm[5:4|9:6]			rs2	10	C.SQSP (*RV128)*
110	uimm[5:2|7:6]			rs2	10	C.SWSP
111	uimm[5:2|7:6]			rs2	10	C.FSWSP (*RV32)*
111	uimm[5:3|8:6]			rs2	10	C.SDSP (*RV64/128)*

Table 16.7: Instruction listing for RVC, Quadrant 2.

Chapter 17

“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 ISA 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.

Chapter 18

“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.

Chapter 19

“T” Standard Extension for Transactional Memory, Version 0.0

This chapter is a placeholder for a future standard extension to provide transactional memory operations.

Despite much research over the last twenty years, and initial commercial implementations, there is still much debate on the best way to support atomic operations involving multiple addresses.

Our current thoughts are to include a small limited-capacity transactional memory buffer along the lines of the original transactional memory proposals.

Chapter 20

“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.

第21章

矢量操作的“ V”标准扩展，版本0.7

当前工作组草案的托管地点为 https://github.com/riscv/riscv-v-spec。

基本矢量扩展旨在为32位指令编码空间内的数据并行执行提供一般支持，而更高版本的矢量扩展则支持某些域的更丰富功能。

第22章

错位原子的“ Zam”标准扩展，v0.1

本章定义了“ Zam”扩展，它通过标准化对未对齐原子存储操作（AMO）的支持来扩展“ A”扩展。在实现“ Zam”的平台上，未对齐的AMO仅需要针对其他访问（包括非原子加载和存储）以原子方式执行对相同地址和相同大小的访问。更准确地说，实现“ Zam”的执行环境应遵循以下原则：

未对准原子的原子性公理 如果 *[R* 和 *w* 配对错位加载并存储来自牡鹿的指令 *H* 具有相同的地址和相同的大小，那么就不会有存储指令 *s* 来自除 *H* 地址和大小与 *[R* 和 *w* 这样，由 *s* 介于由 *[R* 和 *w* 按照全局内存顺序。此外，没有加载指令 *升* 来自除

H 地址和大小与 *[R* 和 *w* 这样由
升 位于由生成的两个内存操作之间 *[R* 或 *w* 按照全局内存顺序。

这种受限形式的原子性旨在平衡需要支持未对准原子的应用程序的需求，以及实现实际提供必要程度原子性的能力之间的平衡。

在“ Zam”下对齐的指令继续像在RVWMO下正常运行一样。

“Zam”的目的是可以通过以下两种方式之一来实现：

1.在本机支持对地址和大小的原子错位访问的硬件上（例如，对于单个高速缓存行中的错位访问）：只需遵循将应用于对齐的AMO的相同规则。

2.在本身不支持对地址和大小的错位访问的硬件上：通过捕获具有该地址和大小的所有指令（包括负载），并在互斥锁内执行它们（通过任何数量的内存操作），给定内存地址和访问大小的函数。可以通过将AMO分成单独的加载和存储操作来模拟AMO，但是所有保留的程序顺序规则（例如，传入和传出语法上的依存关系）必须表现得好像AMO仍然是单个内存操作。

第23章

总商店订购的“ Ztso”标准扩展，v0.1

本章定义了RISC-V总存储订购（RVTSO）内存一致性模型的“ Ztso”扩展名。RVTSO被定义为RVWMO的增量，在本章中进行了定义 14.1 。

Ztso扩展旨在促进最初为x86或SPARC体系结构编写的代码的移植，这两种体系结构默认情况下都使用TSO。它还支持固有提供RVTSO行为并希望将该事实公开给软件的实现。

RVTSO对RVWMO进行了以下调整：

- 所有加载操作的行为就像它们具有acquire-RCpc注释一样
- 所有存储操作的行为就像它们具有release-RCpc注释一样。
- 所有AMO的行为就好像它们都具有Acquisition-RCsc和release-RCsc注释一样。

这些规则将呈现所有PPO规则，但 4 – 7 多余的。它们还使没有同时设置PW和SR的所有非I / O隔离网变得多余。最后，它们还暗示在AMO上任一方向上的存储操作都不会重新排序。

在RVWSO的情况下，与RVWMO一样，存储顺序注释由PPO规则简洁完整地定义 5 – 7 。在这两种内存模型中，

负荷值公理 这允许一个暂存器将值从其存储缓冲区转发到后续的（按程序顺序）加载-也就是说，在其他存储库可见存储之前，可以将它们本地转发。

尽管Ztso并未向ISA添加任何新指令，但假设RVTSO编写的代码将无法在不支持Ztso的实现中正确运行。编译为仅在Ztso下运行的二进制文件应通过二进制文件中的标志进行指示，以便未实现Ztso的平台可以拒绝运行它们。

第24章

RV32 / 64G指令集列表

RISC-V项目的一个目标是将其用作稳定的软件开发目标。为此，我们将基本ISA（RV32I或RV64I）与所选标准扩展名（IMAFD，Zicsr，Zifencei）的组合定义为“通用” ISA，并且对IMAFDZicsr Zifencei指令组合使用缩写G集扩展。本章介绍RV32G和RV64G的操作码映射和指令集列表。

–

实例[4：2] 实例[6：5]	000	001	010	011	100	101	110	111 （> 32 *b*）
00	加载	加载FP *自订0*	MISC-MEM	OP-IMM	AUIPC		OP-IMM-32	48 *b*
01	商店	商店FP *自订1*		AMO	OP	UI	OP-32	64 *b*
10	马德	MSUB	NMSUB	国家发展研究部	FP-FP	*保留的custom-2 / rv128*		48 *b*
11分行		捷豹路虎	*保留的*	日航	系统	*保留的custom-3 / rv128*		≥80 *b*

表24.1：RISC-V基本操作码映射，inst [1：0] = 11

表 24.1 显示了RVG的主要操作码图。设置了3个或更多低位的主要操作码将保留给长度大于32位的指令。标为的操作码 *保留的* 对于自定义指令集扩展，应避免使用它们，因为将来的标准扩展可能会使用它们。主要操作码标记为 *自订0* 和 *自订1* 将在以后的标准扩展中避免使用，建议在基本的32位指令格式内供自定义指令集扩展使用。标有操作码 *自订2 / rv128* 和 *自订3 / rv128* 保留给RV128以后使用，但对于标准扩展将避免使用，因此也可用于RV32和RV64中的自定义指令集扩展。

我们相信RV32G和RV64G可为各种通用计算提供简单而完整的指令集。本章中介绍的可选压缩指令集 16 可以添加（形成RV32GC和RV64GC）以提高性能，代码大小和能效，尽管会增加一些硬件复杂性。

当我们从IMAFDC扩展到进一步的指令集扩展时，添加的指令往往更加特定于域，并且仅对有限的应用程序类别（例如，多媒体或安全性）提供好处。与大多数商业ISA不同，RISC-V ISA设计清楚地将基本ISA和广泛适用的标准扩展与这些更专业的添加分开。章节 25 对将扩展添加到RISC-V ISA的方法进行了更广泛的讨论。

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

RV32I基本指令集

毫米[31:12]				rd	0110111	UI
毫米[31:12]				rd	0010111	联合会
imm [20 | 10：1 | 11 | 19：12]				rd	1101111	日航
imm [11：0]		rs1	000	rd	1100111	捷豹路虎
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	北京大学
imm [11：0]		rs1	000	rd	0000011	磅
imm [11：0]		rs1	001	rd	0000011	LH
imm [11：0]		rs1	010	rd	0000011	轻量级
imm [11：0]		rs1	100	rd	0000011	轻型单位
imm [11：0]		rs1	101	rd	0000011	HU
毫米[11：5]	rs2	rs1	000	imm [4：0]	0100011	SB
毫米[11：5]	rs2	rs1	001	imm [4：0]	0100011	SH
毫米[11：5]	rs2	rs1	010	imm [4：0]	0100011	西南
imm [11：0]		rs1	000	rd	0010011	阿迪
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
调频	pred succ	rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

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

RV64I Base Instruction Set (in addition to RV32I)

imm[11:0]		rs1	110	rd	0000011	LWU
imm[11:0]		rs1	011	rd	0000011	LD
imm[11:5]	rs2	rs1	011	imm[4:0]	0100011	SD
000000	shamt	rs1	001	rd	0010011	SLLI
000000	shamt	rs1	101	rd	0010011	SRLI
010000	shamt	rs1	101	rd	0010011	SRAI
imm[11:0]		rs1	000	rd	0011011	ADDIW
0000000	shamt	rs1	001	rd	0011011	SLLIW
0000000	shamt	rs1	101	rd	0011011	SRLIW
0100000	shamt	rs1	101	rd	0011011	SRAIW
0000000	rs2	rs1	000	rd	0111011	ADDW
0100000	rs2	rs1	000	rd	0111011	SUBW
0000000	rs2	rs1	001	rd	0111011	SLLW
0000000	rs2	rs1	101	rd	0111011	SRLW
0100000	rs2	rs1	101	rd	0111011	SRAW

RV32/RV64 *Zifencei* Standard Extension

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

RV32/RV64 *Zicsr* Standard Extension

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

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

RV64M Standard Extension (in addition to RV32M)

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

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type

RV32A Standard Extension

00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

RV64A Standard Extension (in addition to RV32A)

00010	aq rl		00000	rs1	011	rd	0101111	LR.D
00011	aq rl		rs2	rs1	011	rd	0101111	SC.D
00001	aq rl		rs2	rs1	011	rd	0101111	AMOSWAP.D
00000	aq rl		rs2	rs1	011	rd	0101111	AMOADD.D
00100	aq rl		rs2	rs1	011	rd	0101111	AMOXOR.D
01100	aq rl		rs2	rs1	011	rd	0101111	AMOAND.D
01000	aq rl		rs2	rs1	011	rd	0101111	AMOOR.D
10000	aq rl		rs2	rs1	011	rd	0101111	AMOMIN.D
10100	aq rl		rs2	rs1	011	rd	0101111	AMOMAX.D
11000	aq rl		rs2	rs1	011	rd	0101111	AMOMINU.D
11100	aq rl		rs2	rs1	011	rd	0101111	AMOMAXU.D

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32F Standard Extension

imm[11:0]		rs1	010	rd	0000111	FLW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100111	FSW
rs3	00	rs2	rs1	rm	rd	1000011	FMADD.S
rs3	00	rs2	rs1	rm	rd	1000111	FMSUB.S
rs3	00	rs2	rs1	rm	rd	1001011	FNMSUB.S
rs3	00	rs2	rs1	rm	rd	1001111	FNMADD.S
0000000	rs2	rs1	rm	rd	1010011	FADD.S
0000100	rs2	rs1	rm	rd	1010011	FSUB.S
0001000	rs2	rs1	rm	rd	1010011	FMUL.S
0001100	rs2	rs1	rm	rd	1010011	FDIV.S
0101100	00000	rs1	rm	rd	1010011	FSQRT.S
0010000	rs2	rs1	000	rd	1010011	FSGNJ.S
0010000	rs2	rs1	001	rd	1010011	FSGNJN.S
0010000	rs2	rs1	010	rd	1010011	FSGNJX.S
0010100	rs2	rs1	000	rd	1010011	FMIN.S
0010100	rs2	rs1	001	rd	1010011	FMAX.S
1100000	00000	rs1	rm	rd	1010011	FCVT.W.S
1100000	00001	rs1	rm	rd	1010011	FCVT.WU.S
1110000	00000	rs1	000	rd	1010011	FMV.X.W
1010000	rs2	rs1	010	rd	1010011	FEQ.S
1010000	rs2	rs1	001	rd	1010011	FLT.S
1010000	rs2	rs1	000	rd	1010011	FLE.S
1110000	00000	rs1	001	rd	1010011	FCLASS.S
1101000	00000	rs1	rm	rd	1010011	FCVT.S.W
1101000	00001	rs1	rm	rd	1010011	FCVT.S.WU
1111000	00000	rs1	000	rd	1010011	FMV.W.X

RV64F Standard Extension (in addition to RV32F)

1100000	00010	rs1	rm	rd	1010011	FCVT.L.S
1100000	00011	rs1	rm	rd	1010011	FCVT.LU.S
1101000	00010	rs1	rm	rd	1010011	FCVT.S.L
1101000	00011	rs1	rm	rd	1010011	FCVT.S.LU

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32D Standard Extension

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

RV64D Standard Extension (in addition to RV32D)

1100001	00010	rs1	rm	rd	1010011	FCVT.L.D
1100001	00011	rs1	rm	rd	1010011	FCVT.LU.D
1110001	00000	rs1	000	rd	1010011	FMV.X.D
1101001	00010	rs1	rm	rd	1010011	FCVT.D.L
1101001	00011	rs1	rm	rd	1010011	FCVT.D.LU
1111001	00000	rs1	000	rd	1010011	FMV.D.X

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
rs3	funct2	rs2	rs1	funct3	rd	opcode	R4-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type

RV32Q Standard Extension

imm[11:0]			rs1	100	rd	0000111	FLQ
imm[11:5]		rs2	rs1	100	imm[4:0]	0100111	FSQ
rs3	11	rs2	rs1	rm	rd	1000011	FMADD.Q
rs3	11	rs2	rs1	rm	rd	1000111	FMSUB.Q
rs3	11	rs2	rs1	rm	rd	1001011	FNMSUB.Q
rs3	11	rs2	rs1	rm	rd	1001111	FNMADD.Q
0000011		rs2	rs1	rm	rd	1010011	FADD.Q
0000111		rs2	rs1	rm	rd	1010011	FSUB.Q
0001011		rs2	rs1	rm	rd	1010011	FMUL.Q
0001111		rs2	rs1	rm	rd	1010011	FDIV.Q
0101111		00000	rs1	rm	rd	1010011	FSQRT.Q
0010011		rs2	rs1	000	rd	1010011	FSGNJ.Q
0010011		rs2	rs1	001	rd	1010011	FSGNJN.Q
0010011		rs2	rs1	010	rd	1010011	FSGNJX.Q
0010111		rs2	rs1	000	rd	1010011	FMIN.Q
0010111		rs2	rs1	001	rd	1010011	FMAX.Q
0100000		00011	rs1	rm	rd	1010011	FCVT.S.Q
0100011		00000	rs1	rm	rd	1010011	FCVT.Q.S
0100001		00011	rs1	rm	rd	1010011	FCVT.D.Q
0100011		00001	rs1	rm	rd	1010011	FCVT.Q.D
1010011		rs2	rs1	010	rd	1010011	FEQ.Q
1010011		rs2	rs1	001	rd	1010011	FLT.Q
1010011		rs2	rs1	000	rd	1010011	FLE.Q
1110011		00000	rs1	001	rd	1010011	FCLASS.Q
1100011		00000	rs1	rm	rd	1010011	FCVT.W.Q
1100011		00001	rs1	rm	rd	1010011	FCVT.WU.Q
1101011		00000	rs1	rm	rd	1010011	FCVT.Q.W
1101011		00001	rs1	rm	rd	1010011	FCVT.Q.WU

RV64Q Standard Extension (in addition to RV32Q)

1100011	00010	rs1	rm	rd	1010011	FCVT.L.Q
1100011	00011	rs1	rm	rd	1010011	FCVT.LU.Q
1101011	00010	rs1	rm	rd	1010011	FCVT.Q.L
1101011	00011	rs1	rm	rd	1010011	FCVT.Q.LU

Table 24.2: Instruction listing for RISC-V

Table 24.3 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.

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. Floating-Point Control and Status
0x003	Read/write	fcsr	Register (frm + fflags).
Counters and Timers			
0xC00	Read-only	cycle	Cycle counter for RDCYCLE instruction. Timer for
0xC01	Read-only	time	RDTIME instruction.
0xC02	Read-only	instret	Instructions-retired counter for RDINSTRET instruction. Upper 32 bits of cycle,
0xC80	Read-only	cycleh	RV32I only. Upper 32 bits of time, RV32I only.
0xC81	Read-only	timeh	
0xC82	Read-only	instreth	Upper 32 bits of instret, RV32I only.

Table 24.3: RISC-V control and status register (CSR) address map.

Chapter 25

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.

25.1 Extension Terminology

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

Standard versus Non-Standard Extension

Any RISC-V processor implementation must support a base integer ISA (RV32I, RV32E, RV64I, 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.

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 ISA is 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 25.1 gives the suggested sizes for RISC-V.

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 Major		2^{8}	2^{20}	2^{35}
25-bit	opcode in base 32-bit encoding Quadrant of base		32	2^{17}	2^{32}
30-bit	32-bit encoding		1	2^{12}	2^{27}
32-bit	Minor opcode in 48-bit encoding Major			2^{10}	2^{25}
37-bit	opcode in 48-bit encoding Quadrant of 48-bit			32	2^{20}
40-bit	encoding			4	2^{17}
45-bit	Sub-minor opcode in 64-bit encoding Minor				2^{12}
48-bit	opcode in 64-bit encoding Major opcode in 64-bit				2^{9}
52-bit	encoding				32

Table 25.1: Suggested standard RISC-V instruction encoding space sizes.

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 the 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.

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

Table 25.2: Two-dimensional characterization of standard instruction-set extensions.

Table 25.2 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.

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.

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.

25.2 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 ISA has 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 standardcompatible 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.

25.3 Extensions within fixed-width 32-bit instruction format

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

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

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.

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 24.1), 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 nonstandard 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 24.1).

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.

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.

Other spaces

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

25.4 Adding aligned 64-bit instruction extensions

The simplest approach to provide space for extensions that are too large for the base 32-bit fixedwidth 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 Figure 1.1 . 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.

25.5 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.

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 VLIWaccelerated functions.

Encoded-Length Groups

Another approach is to use the standard length encoding from Figure 1.1 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).

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.

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 ISA lacks 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.

Chapter 26

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.

26.1 Case Sensitivity

The ISA naming strings are case insensitive.

26.2 Base Integer ISA

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

26.3 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 dependences may be implicit in the ISA name: for example, RV32IF is equivalent to RV32IFZicsr, and RV32ID is equivalent to RV32IFD and RV32IFDZicsr.

26.4 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 “RV64I1p0M1p0A1p0F1p0D1p0”, more concisely as “RV64I1M1A1F1D1”.

We introduced the version numbering scheme with the second release. Hence, we define the default version of a standard extension to be the version present at that time, e.g., “RV32I” is equivalent to “RV32I2”.

26.5 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.

26.6 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, IMAFDQLCBJTPVN. For the “Zam” extension for misaligned atomics, for example, the letter “a” indicates the extension is related to the “A” 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”. _ _

Extensions with the “Z” prefix must be separated from other multi-letter extensions by an underscore, e.g., “RV32IMACZicsr Zifencei”. _

26.7 Supervisor-level Instruction-Set Extensions

Standard supervisor-level instruction-set extensions are defined in Volume II, but are named using “S” as a prefix, followed by an alphabetical name and an optional version number. Supervisor-level extensions must be separated from other multi-letter extensions by an underscore.

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

26.8 Hypervisor-level Instruction-Set Extensions

Standard hypervisor-level instruction-set extensions are named like supervisor-level extensions, but beginning with the letter “H” instead of the letter “S”.

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

26.9 Machine-level Instruction-Set Extensions

Standard machine-level instruction-set extensions are prefixed with the three letters “Zxm”.

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

26.10 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. They 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.

26.11 Subset Naming Convention

Table 26.1 summarizes the standardized extension names.

Subset	Name	Implies
Base ISA		
Integer	I	
Reduced Integer	E	
Standard Unprivileged Extensions		
Integer Multiplication and Division	M	
Atomics	A	
Single-Precision Floating-Point	F	Zicsr
Double-Precision Floating-Point	D	F
General	G	IMADZifencei
Quad-Precision Floating-Point	Q	D
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	
Standard Supervisor-Level Extensions		
Supervisor-level extension “def”	Sdef	
Standard Hypervisor-Level Extensions		
Hypervisor-level extension “ghi”	Hghi	
Standard Machine-Level Extensions		
Machine-level extension “jkl”	Zxmjkl	
Non-Standard Extensions		
Non-standard extension “mno”	Xmno	

Table 26.1: Standard ISA 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.

Chapter 27

History and Acknowledgments

27.1 “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 [3], 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 [13]. 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 RISCV Foundation website at https://riscv.org contains the latest information on the Foundation membership and various open-source projects using RISC-V.

27.2 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 T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. T0 was a vector processor based on the MIPS-II ISA, with Krste Asanovi´ć 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 T0 ISA design, particularly supervisor mode, and to the manual text. John Hauser also provided considerable feedback on the T0 ISA design.

The Scale (Software-Controlled Architecture for Low Energy) project at MIT, begun in 2000, built upon the T0 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 T0 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 Asanovi´ć 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 Asanovi´ć 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 Asanovi´ć, and John Lazzaro.

The Maven (Malleable Array of Vector-thread ENgines) project was a second-generation vectorthread 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 Avi˘zienis 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 Asanovi´c, 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.

27.3 History from Revision 2.0 of ISA manual

Multiple implementations of RISC-V processors have been completed, including several silicon fabrications, as shown in Figure 27.1 .

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 EOS22	RV64G1p99 Xhwacha2
	March 7, 2014	IBM 45nm SOI	RV64G1p9999 Xhwacha3

Table 27.1: Fabricated RISC-V testchips.

The first RISC-V processors to be fabricated were written in Verilog and manufactured in a preproduction 28 nm FDSOI technology from ST as the Raven-1 testchip in 2011. Two cores were developed by Yunsup Lee and Andrew Waterman, advised by Krste Asanovi´c, 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 Avi˘zienis, and Yunsup Lee, advised by Krste Asanovi´c, 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 28 nm FDSOI (Raven-2, Raven-3), and five times in IBM 45 nm 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 Asanovi´ć. EOS16–EOS22 chips include dual cores with a cache-coherence protocol developed by Henry Cook and Andrew Waterman, advised by Krste Asanovi´ć. EOS14 silicon has successfully run at 1.25GHz. EOS16 silicon suffered from a bug in the IBM pad libraries. EOS18 and EOS20 have successfully run at 1.35GHz.

Contributors to the Raven testchips include Yunsup Lee, Andrew Waterman, Rimas Avi˘žienis, Brian Zimmer, Jaehwa Kwak, Ruzica Jevti´ć, Milovan Blagojevi´ć, Alberto Puggelli, Steven Bailey, Ben Keller, Pi-Feng Chiu, Brian Richards, Borivoje Nikoli´ć, and Krste Asanovi´ć.

Contributors to the EOS testchips include Yunsup Lee, Rimas Avi˘žienis, AndrewWaterman, Henry Cook, Huy Vo, Daiwei Li, Chen Sun, Albert Ou, Quan Nguyen, Stephen Twigg, Vladimir Stojanovi´ć, and Krste Asanovi´ć.

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.

AndrewWaterman completed a Master’s thesis with a preliminary design of the RISC-V compressed instruction set [23].

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 Avi˘žienis, and Krste Asanovi´ć.

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-ofOrder 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.

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.

27.4 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.

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.

27.5 History from Revision 2.2

Acknowledgments

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

27.6 History for Revision 2.3

Uptake of RISC-V continues at breakneck pace.

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

Daniel Lustig, Arvind, Krste Asanovi´ć, 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.

27.7 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 HR0011-12-2-0016. DARPA POEM program Award HR0011-11-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.

Appendix A

RVWMO Explanatory Material, Version 0.1

This section provides more explanation for RVWMO (Chapter 14), 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 14 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 RISCV 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. Figure A.1 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. Each figure shows the litmus test code on the left, and a visualization of one particular valid or invalid execution on the right.

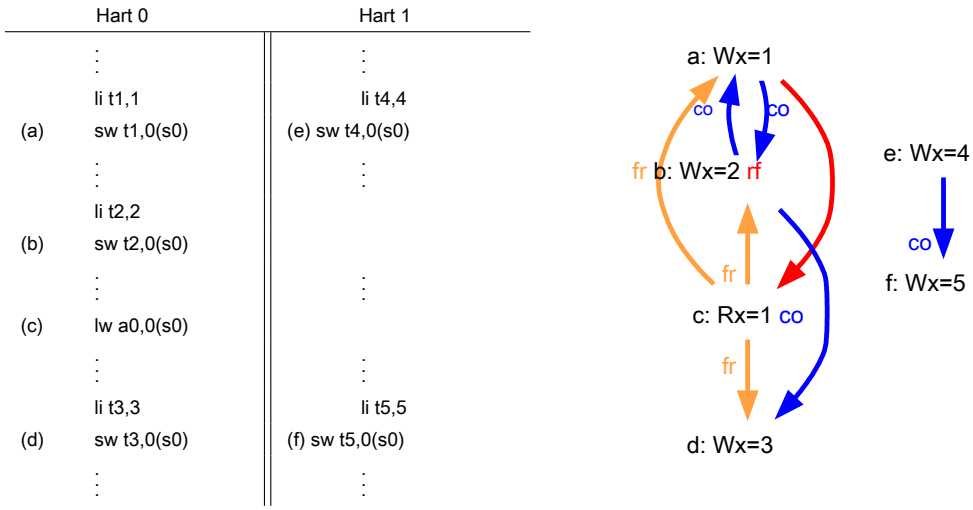


Figure A.1: A sample litmus test and one forbidden execution (a0=1).

Litmus tests are used to understand the implications of the memory model in specific concrete situations. For example, in the litmus test of Figure A.1 , 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.

The diagram shown to the right of each litmus test shows a visual representation of the particular execution candidate being considered. These diagrams use a notation that is common in the memory model literature for constraining the set of possible global memory orders that could produce the execution in question. It is also the basis for the herd models presented in Appendix B.2 . This

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

Table A.1: A key for the litmus test diagrams drawn in this appendix

notation is explained in Table A.1 . 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 Figure A.1 , 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 1 . 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 https://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

> 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

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 litmus test of Figure A.2 . 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:

- (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

	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

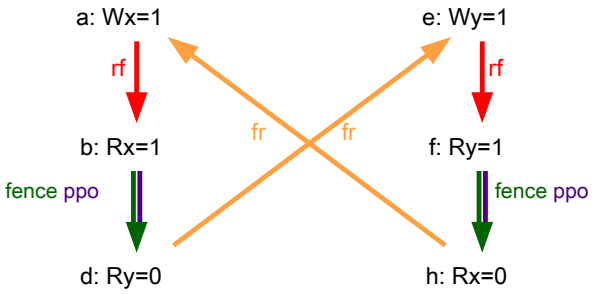


Figure A.2: A store buffer forwarding litmus test (outcome permitted)

- (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).

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

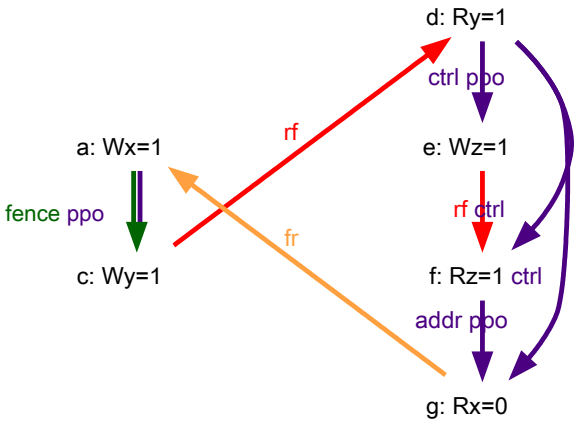


Figure A.3: The “PPOCA” store buffer forwarding litmus test (outcome permitted)

Another test that highlights the behavior of store buffers is shown in Figure A.3 . 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 a0, 0(s0) (b) sd t1, 0(s0) (c) sc.d t2, 0(s0)	(a) lr.d a0, 0(s0) (b) sw t1, 4(s0) (c) sc.d t2, 0(s0)	(a) lr.w a0, 0(s0) (b) sw t1, 4(s0) (c) sc.w t2, 0(s0)	(a) lr.w a0, 0(s0) (b) sw t1, 4(s0) (c) sc.w t2, 8(s0)

Figure A.4: In all four (independent) code snippets, the store-conditional (c) 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 Figure A.4 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 Figure A.4 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 Section A.3.2 , this reflects the observable behavior of implementations that forward values from buffered stores to later loads.

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

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

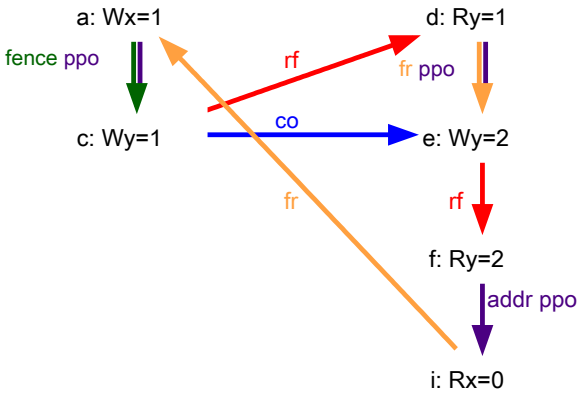


Figure A.5: Litmus test MP+fence.w.w+fri-rfi-addr (outcome permitted)

Consider the litmus test of Figure A.5 , 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.

Consider the litmus test of Figure A.6 . 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

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

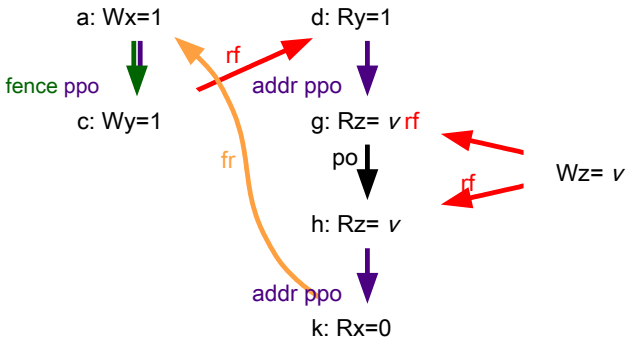


Figure A.6: Litmus test RSW (outcome permitted)

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 Figure A.6 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, PPO rule 2 forbids this CoRR violation from occurring. As such, PPO 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: PPO 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, PPO 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 cumulativity” 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.

```
        sd          x1, (a1)          # Arbitrary unrelated store
        ld          x2, (a2)          # Arbitrary unrelated load
        li          t0, 1             # Initialize swap value.
again:
        amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock. bnez
                    t0, again         # Retry if held.
        # . . .
        # Critical section.
        # . . .
        amoswap.w.rl x0, x0, (a0) # Release lock by storing 0. sd
                    x3, (a3)          # Arbitrary unrelated store
        ld          x4, (a4)          # Arbitrary unrelated load
```

Figure A.7: A spinlock with atomics

Consider Figure A.7 . Because this example uses *aq,* the loads and stores in the critical section are guaranteed to appear in the global memory order after the AMOSWAP used to acquire the lock. However, assuming a0, a1, and a2 point to different memory locations, the loads and stores in the critical section may or may not appear after the “Arbitrary unrelated load” at the beginning of the example in the global memory order.

```
        sd          x1, (a1)          # Arbitrary unrelated store
        ld          x2, (a2)          # Arbitrary unrelated load
        li          t0, 1             # Initialize swap value.
again:
        amoswap.w   t0, t0, (a0) # Attempt to acquire lock.
        fence       r, rw             # Enforce "acquire" memory ordering
        bnez        t0, again         # Retry if held.
        # . . .
        # Critical section.
        # . . .
        fence       rw, w             # Enforce "release" memory ordering
        amoswap.w   x0, x0, (a0) # Release lock by storing 0. x3, (a3)
        sd                            # Arbitrary unrelated store
        ld          x4, (a4)          # Arbitrary unrelated load
```

Figure A.8: A spinlock with fences

Now, consider the alternative in Figure A.8 . 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 Figure A.8 , not by the *rl* in Figure A.7 .

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-releaseto-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 Section A.5

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 14.1 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.*

```
(a) fadd f3,f1,f2
(b) fadd f6,f4,f5
(c) csrrs a0,fflags,x0
```

Figure A.9: (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 Figure A.9 , (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.

```
ld a1,0(s0)
xor a2,a1,a1
add s1,s1,a2
ld a5,0(s1)
```

Figure A.10: 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 Figure A.10 , 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 nondependent 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.

```
        lw x1,0(x2)
        bne x1,x0,next
        sw x3,0(x4)
next:   sw x5,0(x6)
```

Figure A.11: 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 Figure A.11 : 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.

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

Figure A.12: Another syntactic control dependency

Likewise, consider Figure A.12 . 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.

Initial values: 0(s0)=1; 0(s1)=1

	Hart 0	Hart 1
(a)	ld a0,0(s0)	(e) ld a3,0(s2)
(b)	lr a1,0(s1)	(f) sd a3,0(s0)
(c)	sc a2,a0,0(s1)	
(d)	sd a2,0(s2)	

Outcome: a0=0, a3=0

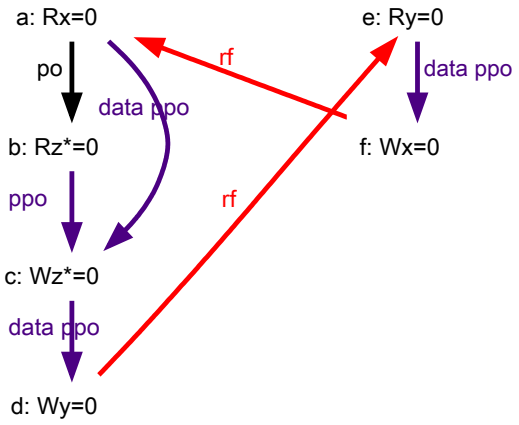


Figure A.13: A variant of the LB litmus test (outcome forbidden)

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 Figure A.13 . 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 14.3 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*

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

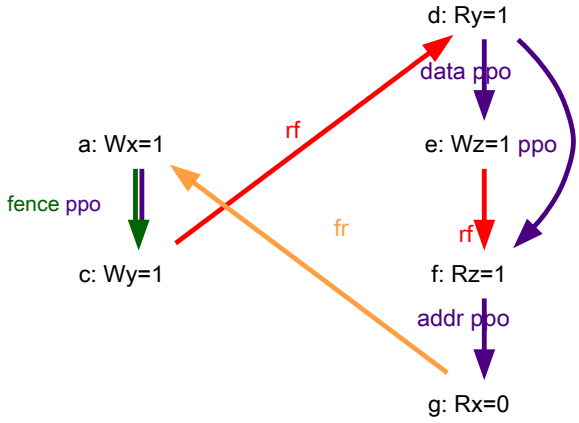


Figure A.14: Because of PPO rule 12 and the data dependency from (d) to (e), (d) must also precede (f) in the global memory order (outcome forbidden)

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 Figure A.14 : (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.

If there were another store to the same address in between (e) and (f), as in Figure A.15 , 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

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

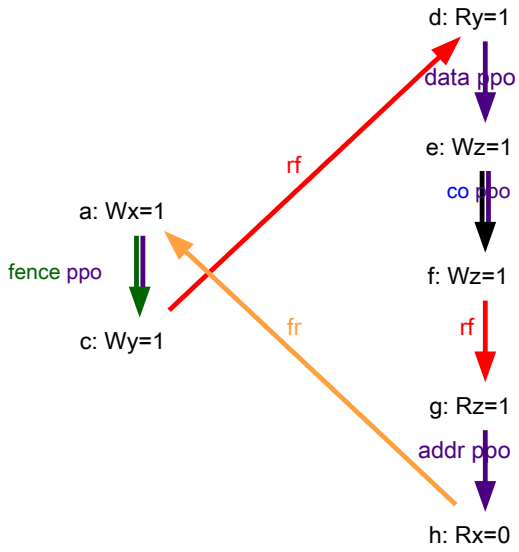


Figure A.15: Because of the extra store between (e) and (g), (d) no longer necessarily precedes (g) (outcome permitted)

to failed address resolution, and in this sense, rule 13 can be seen as somewhat of a special case of rule 11 .

Hart 0	Hart 1
	li t1, 1
(a) lw a0,0(s0)	(d) lw a1, 0(s1) (e) lw a2,
(b) fence rw,rw	0(a1) (f) sw t1, 0(s0)
(c) sw s2,0(s1)	

Outcome: a0=1, a1=t

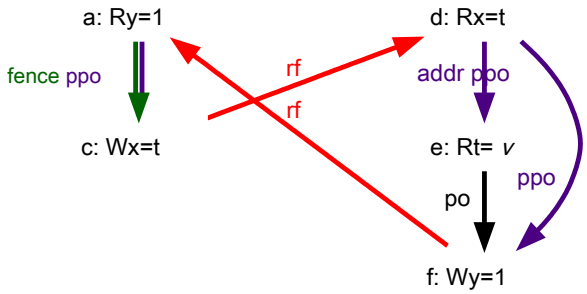


Figure A.16: Because of the address dependency from (d) to (e), (d) also precedes (f) (outcome forbidden)

Consider Figure A.16 : (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/Omemory 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 noncoherent 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 14 , 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.

```
sd t0, 0(a0)
fence w,o
sd a0, 0(a1)
```

Figure A.17: Ordering memory and I/O accesses

When a fence is in fact used, implementations must assume that the device may attempt to access memory immediately after receiving the MMIO signal, and subsequent memory accesses from that device to memory must observe the effects of all accesses ordered prior to that MMIO operation. In other words, in Figure A.17 , suppose 0(a0) is in main memory and 0(a1) is the address of a device register in I/O memory. If the device accesses 0(a0) upon receiving the MMIO write, then that load must conceptually appear after the first store to 0(a0) according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and main memory accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

Many architectures include separate notions of “ordering” and “completion” fences, especially as it relates to I/O (as opposed to regular main memory). Ordering fences simply ensure that memory operations stay in order, while completion fences ensure that predecessor accesses have all completed before any successors are made visible. RISC-V does not explicitly distinguish between ordering and completion fences. Instead, this distinction is simply inferred from different uses of the FENCE bits.

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices and DMA operations are required to access memory coherently and via strongly-ordered I/O channels. Therefore, accesses to regular main memory regions that are concurrently accessed by external devices can also use the standard synchronization mechanisms. Implementations that do not conform to the Unix Platform Specification and/or in which devices do not access memory coherently will need to use mechanisms (which are currently platform-specific or device-specific) to enforce coherency.

I/O regions in the address space should be considered non-cacheable regions in the PMAs for those regions. Such regions can be considered coherent by the PMA if they are not cached by any agent.

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe) may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

A.5 Code Porting and Mapping Guidelines

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

Table A.2: Mappings from TSO operations to RISC-V operations

Table A.2 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 A.2 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.

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

Table A.3: Mappings from Power operations to RISC-V operations

Table A.3 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.

ARM Operation	RVWMO Mapping
Load	l{b|h|w|d}
Load-Acquire	fence rw, rw; l{b|h|w|d}; fence r,rw lr.{w|d}
Load-Exclusive	
Load-Acquire-Exclusive	lr.{w|d}.aqrl
Store	s{b|h|w|d}
Store-Release	fence rw,w; s{b|h|w|d}
Store-Exclusive	sc.{w|d}
Store-Release-Exclusive	sc.{w|d}.rl
dmb	fence rw,rw
dmb.ld	fence r,rw
dmb.st	fence w,w
isb	fence.i; fence r,r

Table A.4: Mappings from ARM operations to RISC-V operations

Table A.4 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 A.4 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 A.5 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 FENCEW,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 noncoherent DMAmay 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.

Linux Operation	RVWMO Mapping
smp_mb()	fence rw,rw
smp_rmb()	fence r,r
smp_wmb()	fence w,w
dma_rmb()	fence r,r
dma_wmb()	fence w,w
mb()	fence iorw,iorw
rmb()	fence ri,ri
wmb()	fence wo,wo
smp_load_acquire()	l{b|h|w|d}; fence r,rw
smp_store_release()	fence.tso; s{b|h|w|d}

Linux Construct	RVWMO AMO Mapping
atomic <op> relaxed	amo<op>.{w|d}
atomic <op> acquire	amo<op>.{w|d}.aq
atomic <op> release	amo<op>.{w|d}.rl
atomic <op>	
	amo<op>.{w|d}.aqrl

Linux Construct	RVWMO LR/SC Mapping
atomic <op> relaxed	loop: lr.{w|d}; <op>; sc.{w|d}; bnez loop
atomic <op> acquire	loop: lr.{w|d}; <op>; sc.{w|d}.aq; bnez loop
atomic <op> release	fence.tso; loop: lr.{w|d}; <op>; sc.{w|d}; bnez loop
atomic <op>	loop: lr.{w|d}.aqrl; <op>; sc.{w|d}.aqrl; bnez loop OR fence.tso; loop: lr.{w|d}; <op>; sc.{w|d}.aq; bnez loop

Table A.5: Mappings from Linux memory primitives to RISC-V primitives. 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.

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 A.5 .

```
                                        RVWMO Mapping:
                                        (a) lw          a0, 0(s0)
Linux code:                             (b) fence.tso   // vs. fence rw,w
(a) int r0 = *x;                        (c) sd          x0,0(s1)
(bc) spin_unlock(y, 0);                     . . .
        . . .                               loop:
        . . .                           (d) amoswap.d.aq a1,t1,0(s1)
(d)     spin_lock(y);                       bnez        a1,loop
(e)     int r1 = *z;                    (e) lw          a2,0(s2)
```

Figure A.18: Orderings between critical sections in Linux

For example, the critical section ordering rule currently being debated by the Linux community would require (a) to be ordered before (e) in Figure A.18 . If that will indeed be required, then it would be insufficient for (b) to map as FENCE RW,W. That said, these mappings are subject to change as the Linux Kernel Memory Model evolves.

C/C++ Construct	RVWMO Mapping
Non-atomic load	l{b|h|w|d}
atomic load(memory order relaxed) atomic	l{b|h|w|d}
load(memory order acquire) atomic load(memory	l{b|h|w|d}; fence r,rw
order seq cst)	fence rw,rw; l{b|h|w|d}; fence r,rw
Non-atomic store	s{b|h|w|d}
atomic store(memory order relaxed) atomic	s{b|h|w|d}
store(memory order release) atomic store(memory	fence rw,w; s{b|h|w|d}
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) atomic	amo<op>.{w|d}
<op>(memory order acquire) atomic <op>(memory	amo<op>.{w|d}.aq
order release) atomic <op>(memory order acq rel)	amo<op>.{w|d}.rl
atomic <op>(memory order seq cst)	amo<op>.{w|d}.aqrl
	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 A.6: Mappings from C/C++ primitives to RISC-V primitives.

Table A.6 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 A.7 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.

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

C/C++ Construct	RVWMO Mapping
Non-atomic load	l{b|h|w|d}
atomic load(memory order relaxed) atomic	l{b|h|w|d}
load(memory order acquire) atomic load(memory	l{b|h|w|d}.aq
order seq_cst)	l{b|h|w|d}.aq
Non-atomic store	s{b|h|w|d}
atomic store(memory order relaxed) atomic	s{b|h|w|d}
store(memory order release) atomic store(memory	s{b|h|w|d}.rl
order seq_cst)	s{b|h|w|d}.rl
atomic thread fence(memory order acquire) fence r,rw atomic thread fence(memory	
order release) fence rw,w atomic thread fence(memory order acq_rel) fence.tso atomic	
thread fence(memory_order_seq_cst) fence rw,rw	

C/C++ Construct	RVWMO AMO Mapping
atomic <op>(memory order relaxed) atomic	amo<op>.{w|d}
<op>(memory order acquire) atomic <op>(memory	amo<op>.{w|d}.aq
order release) atomic <op>(memory order acq_rel)	amo<op>.{w|d}.rl
atomic <op>(memory order seq_cst)	amo<op>.{w|d}.aqrl
	amo<op>.{w|d}.aqrl

C/C++ Construct	RVWMO LR/SC Mapping
atomic <op>(memory order relaxed) atomic	lr.{w|d}; <op>; sc.{w|d}
<op>(memory order acquire) atomic <op>(memory	lr.{w|d}.aq; <op>; sc.{w|d}
order release) atomic <op>(memory order acq_rel)	lr.{w|d}; <op>; sc.{w|d}.rl
atomic <op>(memory order seq_cst)	lr.{w|d}.aq; <op>; sc.{w|d}.rl
	lr.{w|d}.aq *; <op>; sc.{w|d}.rl

* must be lr.{w|d}.aqrl in order to interoperate with code mapped per Table A.6

Table A.7: Hypothetical mappings from C/C++ primitives to RISC-V primitives, if native loadacquire and store-release opcodes are introduced.

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.

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/multiplereader 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 highperformance 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:

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

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)

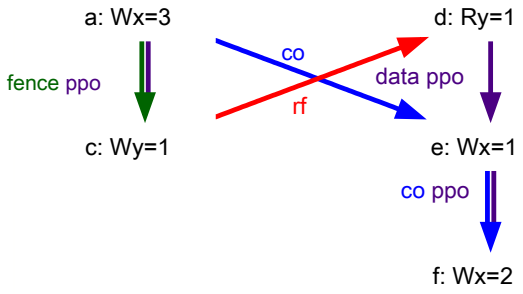


Figure A.19: Write subsumption litmus test, allowed execution.

- (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
- A transactional memory subset of the 'T' ISA extension
- '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

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 a2,0(s0)

Outcome: a0=1, a1=1, a2=0, a3=1, a4=0

Figure A.20: 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 a2,0(s0)

Outcome: a0=0, a1=1, a2=0

Figure A.21: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

Hart 0		Hart 1
li t1, 1		li t1, 1
(a) lw a0,0(s0)	(d)	sw t1,4(s1)
(b) fence rw,rw	(e)	ld a1,0(s1)
(c) sw t1,0(s1)	(f)	lw a2,4(s1)
		xor a3,a2,a2
		add s0,s0,a3
	(g)	sw a2,0(s0)

Outcome: a0=1, a1=0x100000001, a1=1

Figure A.22: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

A.7 Known Issues

A.7.1 Mixed-size RSW

There is a known discrepancy between the operational and axiomatic specifications within the family of mixed-size RSW variants shown in Figures A.20 – A.22 . To address this, we may choose to add something like the following new PPO rule: Memory operation *a* precedes memory operation *b*
in preserved program order (and hence also in the global memory order) if *a* precedes *b* in program order, *a* and *b* both access regular main memory (rather than I/O regions), *a* is a load, *b* is a store, there is a load *m* between *a* and *b,* there is a byte *x* that both *a* and *m* read, there is no store between *a* and *m* that writes to *x,* and *m* precedes *b* in PPO. In other words, in herd syntax, we may choose to add “(po-loc & rsw);ppo;[W] ” to PPO. Many implementations will already

enforce this ordering naturally. As such, even though this rule is not official, we recommend that implementers enforce it nevertheless in order to ensure forwards compatibility with the possible future addition of this rule to RVWMO.

Appendix B

Formal Memory Model Specifications, Version 0.1

To facilitate formal analysis of RVWMO, this chapter presents a set of formalizations using different tools and modeling approaches. Any discrepancies are unintended; the expectation is that the models describe exactly the same sets of legal behaviors.

This appendix should be treated as commentary; all normative material is provided in Chapter 14 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 (http://alloy.mit.edu). This model is available online at https://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 }
```

Figure B.1: 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 // such that x is from a different hart ,
            x not in same_hart[r] and x in r.~rf.^gmo
            and r.pair in x.^gmo                            // x follows (the store r reads from) in gmo , // and r follows x in gmo

}

// Progress Axiom implicit: Alloy only considers finite executions

pred RISCV_mm { LoadValue and Atomicity /* and Progress */ }
```

Figure B.2: 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) }
```

Figure B.3: 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 { ^po :> FencePRSR ).(^po :>
        (Load          <:    Load) ^po :> FencePRSW ).(^po :> Store) ^po :> FencePWSR
    + (Load            <:    ).(^po :> Load) ^po :> FencePWSW ).(^po :> Store)
    + (Store <:
    + (Store <:
    + (Load            <:    ^po :> FenceTSO) .(^po :> MemoryEvent) ^po :> FenceTSO) .(^po :> Store)
    + (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 }
```

Figure B.4: 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 , // subtract pairs that read from the same store , //
    - (~rf.rf)                                  and subtract out "fri -rfi" patterns
    - (po_loc.rfi)
}

// 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]

}
```

Figure B.5: The RVWMO memory model formalized in Alloy (5/5: Auxiliaries)

B.2牛群中的正式公理规范

工具 一群 将内存模型和石蕊测试作为输入，并在内存模型之上模拟测试的执行。内存模型以领域特定语言编写

猫。 本节提供两个 猫 RVWMO的内存模型。第一个模型，图 B.7 ，跟随 *全局内存顺序* 章节 14 ，RVWMO的定义，对于

猫 模型。第二个模型，图 B.8 是一个等效的，更有效的，基于偏序的RVWMO模型。

模拟器 一群 是 y 工具套件—请参阅 http://diy.inria.fr 用于软件和文档。可以在以下网址在线获取更多型号和型号 http://diy.inria.fr/cats7/riscv/
。

```
(**********)
(*公用事业*)
(**********)

(*所有围栏关系*)
让 fence.rr = [R]; fencerel(Fence.rr); [R]
让 fence.rw = [R]; fencerel(Fence.rw); [W]
让 fence.r.rw = [R]; fencerel(Fence.r.rw); [M]
让 fence.wr = [W]; fencerel(Fence.wr); [R]
让 fence.ww = [W]; fencerel(Fence.ww); [W]
让 fence.w.rw = [W]; fencerel(Fence.w.rw); [M]
让 fence.rw.r = [M]; fencerel(Fence.rw.r); [R]
让 fence.rw.w = [M]; fencerel(Fence.rw.w); [W]
让 fence.rw.rw = [M]; fencerel(Fence.rw.rw); [M]
让 fence.tso =
    让 f = ([W]; f; [W])中的fencerel(Fence.tso) | ([R]; f; [M])


让 围栏=
    fence.rr | fence.rw | fence.r.rw | fence.wr | fence.ww | fence.w.rw | fence.rw.r | f
    ence.rw.w | fence.rw.rw | 围栏



(*同一地址， 没有 W到*之间的相同地址
让 po-loc- 没有 -w = po-loc \ (po-loc? ; [W]; po-loc) (*读相同的写*)

让 rsw = rf ^ -1; rf
(*获得，或更强                                    *)
让 AQ = Acq | AcqRel
(*发布或更强*)，RL = RelAcqRel

(*所有RCsc *)
让 RCsc = Acq | Rel | AcqRel
(* Amo事件既是R也是W，关系rmw是成对的lr / sc *)
让 AMO =读写
让 StCond =范围(rmw)

(**********)
(* ppo规则*)
(**********)

(*重叠-地址排序*)
让 r1 = [M]; po-loc; [W]和r2 = ([R]; po-loc- 没有 -w; [R]) \ rsw和r3 = [AM
O | StCond]; rfi; [R] (*显式同步*)和r4 =围栏



并且r5 = [AQ]; po; [M]和r6 = [M]; po; [RL
]
并且r7 = [RCsc]; po; [RCsc]和r8 = rmw

(*语法依赖性*)和r9 = [M]; addr; [M]

并且r10 = [M]; data; [W]和r11 = [M]; ctrl; [W]

(*管道依赖项*)
和r12 = [R]; (addr | data); [W]; rfi; [R]和r13 = [R]; addr; [M]; po; [W]


让 ppo = r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | 1313
```

图B.6： riscv-defs.cat， 保留的程序顺序的群组定义（1/3）

```
总

(*请注意，畜群已定义了自己的RF关系*)

(*定义ppo *)
包括“ riscv -defs.cat”

(************************************)
(*生成全局内存顺序*)(************************************)


让 gmo0 = (*前驱：即，将gmo作为包含gmo0 *的总订单构建)
    loc＆(W \ FW) * FW | #在写入相同位置ppo之后最终写入|
                                        #ppo兼容
    射频                                #包括畜群外部RF(优化)

(*走过去 所有 gmo0 *)的线性扩展
            来自线性化的gmo(M \ IW，gmo0)

(*预先添加初始写入-方便 对于 计算rfGMO *)
让 gmo = gmo | loc＆IW *(M \ IW)

(**********)
(*公理*)
(**********)

(*根据负载值公理计算rf，又名rfGMO *)
让 WR = loc＆([W]; (gmo | po); [R])
让 rfGMO = WR \ (loc＆([W]; gmo); WR)

(*检查群rf和rfGMO是否相等*)空(rf \ rfGMO) | (rfGMO \ rf)作为RfCons


(*原子性公理*)
让 infloc = (gmo＆loc)^-1
让 inflocext = infloc和ext
让 温赛德                    = (infloc; rmw; inflocext)＆(infloc; rf; rmw; inflocext)＆[W]
空winside作为原子
```

图B.7： riscv.cat， RVWMO内存模型的畜群版本(2/3)

部分的

(**********)
(*定义*)
(**********)

(*定义ppo *)
包括“ riscv -defs.cat”

(*计算一致性关系*)包括“ cos -opt.cat”

(**********)
(*公理*)
(**********)

(*每个位置的Sc *)
非循环co | rf | fr | po-loc作为相干性

(*主模型公理*)
非循环co | rfe | fr | ppo作为模型

(*原子性公理*)
将rmw＆(fre; coe)作为原子

图B.8： riscv.cat， RVWMO内存模型的另一种显示方式(3/3)

B.3操作记忆模型

这是RVWMO内存模型以操作方式的替代表示。它旨在接纳与公理表达完全相同的扩展行为：对于任何给定的程序，仅当公理表达允许时才允许执行。

公理表达被定义为完成候选者的谓词。相比之下，此操作表示具有抽象的微体系结构风格：表示为状态机，状态表示硬件机状态的抽象表示，并具有明确的乱序和推测性执行（但是从更多实现中抽象出来）特定的微体系结构详细信息，例如寄存器重命名，存储缓冲区，高速缓存层次结构，高速缓存协议等）。这样，它可以提供有用的直觉。它还可以渐进地构造执行，从而可以交互地，随机地探索较大示例的行为，而公理模型则需要完整的候选执行，可以检查公理。

该操作演示涵盖混合大小的执行，以及具有不同的2的幂的字节大小的潜在重叠内存访问。错误对齐的访问分为单字节访问。

操作模型，以及RISC-V ISA语义的片段（RV64I和A），
集成到 rmem 探索工具（ https://github.com/rems-project/rmem ）。rmem
可以探索石蕊测试（请参阅 A2 ）和小型ELF二进制文件，它们是完全，伪随机和交互式的。在 rmem， ISA语义在Sail中明确表达（请参见 https://github.com/
rems-project / sail 航行语言，以及 https://github.com/rems-project/sail-riscv
对于RISC-V ISA模型），并发语义以Lem表示（请参见 https :
//github.com/rems-project/lem 对于Lem语言）。

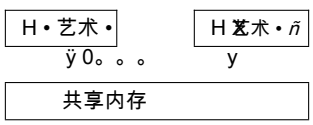
rmem 具有命令行界面和Web界面。Web界面完全在客户端运行，并与石蕊测试库一起在线提供： http://www.cl.cam。

ac.uk/~pes20/rmem 。 命令行界面比Web界面快，特别是在穷举模式下。

以下是模型状态和转换的非正式介绍。正式模型的描述从下一部分开始。

术语：与公理表示法不同，这里的每个内存操作都是加载或存储。因此，AMO引起两个不同的内存操作，即加载和存储。当与“指令”结合使用时，术语“加载”和“存储”是指引起此类存储器操作的指令。因此，两者都包含AMO指令。术语“获取”指的是带有acquire-RCpc或acquiredRCsc注释的指令（或其存储操作）。术语“释放”是指带有release-RCpc或release-RCsc批注的指令（或其存储操作）。

模型状态 模型状态包括 X 共享内存和哈特状态的元组。



共享内存状态以它们传播的顺序记录了到目前为止已经传播的所有存储器存储操作（这可以提高效率，但是为了表示的简单起见，我们将其保持这种方式）。

每个哈特状态主要由指令实例树组成，其中一些实例已经
完成，其中一些还没有。未完成的指令实例可以服从 *重新开始*，
例如，如果它们依赖的是杂乱无章的或推测性的负载，结果却是不合理的。

条件分支指令和间接跳转指令在指令树中可能具有多个后继指令。完成此类指令后，所有未采用的替代路径都将被丢弃。

指令树中的每个指令实例都具有一个状态，该状态包括指令内语义（此指令的ISA伪代码）的执行状态。该模型使用Sail中指令内语义的形式化。可以将指令的执行状态视为伪代码控制状态，伪代码调用堆栈和局部变量值的表示。指令实例状态还包括有关实例的内存和寄存器占用空间，其寄存器的读写操作，其内存操作，是否完成等信息。

模型转换 模型为任何模型状态定义了一组允许的过渡，每个过渡都是到新的抽象机状态的单个原子步骤。一条指令的执行通常会涉及许多转换，并且它们可能在操作模型执行中与其他指令产生的转换交织在一起。每个转换都来自单个指令实例。它会更改该实例的状态，并且可能依赖于或更改其他暂存器状态和共享内存状态，但是它不依赖于其他暂存器状态，也不会更改它们。过渡在下面介绍，并在章节中定义 B.3.5 ，每个条件都有一个前提条件和一个过渡后模型状态的构造。

所有指令的转换：

- 取指令 ：此转换表示作为先前获取的指令实例（或初始获取地址）的程序顺序后继程序的新指令实例的获取和解码。

  该模型假定指令存储器是固定的。它没有描述自我修改代码的行为。特别是 取指令 过渡不会生成内存加载操作，并且共享内存不参与过渡。取而代之的是，该模型依赖于外部oracle，该Oracle在给定内存位置时会提供操作码。

  - 注册写 ：这是对寄存器值的写入。
  - 注册阅读 ：这是从写入该寄存器的最新程序顺序前驱指令实例读取的寄存器值。
  - 伪代码内部步骤 ：这涵盖了伪代码内部计算：算术，函数调用等。
  - 完成指令 ：至此，指令伪代码完成，指令无法重新启动，内存访问无法被放弃，并且所有的内存影响都已发生。对于条件转移指令和间接跳转指令，所有

从不是写到地址的地址获取 *个人电脑* 寄存器连同它们下面的指令实例的子树一起被丢弃。

特定于加载指令的转换：

- 启动内存加载操作 ：在这一点上，加载指令的内存占用是暂时已知的（如果重新启动较早的指令，它可能会更改），并且可以开始满足其单独的内存加载操作。

- 通过从未传播的存储转发来满足内存加载操作 ：通过转发以前来自程序顺序的内存存储操作，这部分或全部满足了单个内存加载操作。

- 通过内存满足内存加载操作 ：这完全满足了内存中单个内存加载操作的出色要求。

- 完成加载操作 ：至此，指令的所有存储器加载操作已完全满足，指令伪代码可以继续执行。加载指令可以重新启动，直到 完成指令 过渡。但是，在某些情况下，模型可能甚至在加载指令完成之前就将其视为不可重启（例如，参见 传播商店运营 ）。

特定于存储指令的转换：

- 启动内存存储区操作足迹 ：至此，存储的内存占用量是暂时已知的。

- 实例化内存存储操作值 ：此时，内存存储操作具有其值，并且可以通过从它们进行转发来满足程序顺序后继的内存加载操作。

- 提交商店指示 ：在这一点上，保证可以执行存储操作（该指令不再可以重新启动或丢弃），并且可以开始传播到内存中。

- 传播商店运营 ：这会将单个内存存储操作传播到内存。
- 完成商店运营 ：此时，指令的所有内存存储操作都已传播到内存，并且指令伪代码可以继续执行。

特定于的转换 SC 说明：

- 早 SC 失败 ：这导致 SC 失败，要么是自发失败，要么是因为它未与先前的程序顺序配对 先生

- 已配对 SC ： 此过渡表明 SC 与一个配对 r 并可能成功。

- 提交和传播存储的操作 SC ： 这是过渡的原子执行
  提交商店指示 和 传播商店运营 ，只有在 r 从中读取尚未被覆盖。

- 晚了 SC 失败 ：这导致 SC 失败，要么是自发失败，要么是因为 r 读取的内容已被覆盖。

特定于AMO指令的过渡：

- 满足，提交和传播AMO的操作 ：这是满足装入操作，执行所需的算术和传播存储操作所需的所有转换的原子执行。

栅栏指令特有的转换：

- 提交围栏

标记为过渡的过渡。只要满足他们的先决条件，就总是可以急切地采取行动，而不排除其他行为；的 •不能。虽然 取指令 标有 •，只要不被无限次使用，就可以积极地使用它。

提取非AMO加载指令的实例后，通常将按以下顺序进行以下转换：

1。 注册阅读

2。 启动内存加载操作

3。 通过从未传播的存储转发来满足内存加载操作 和/或 满足记忆
从内存进行装载操作 （为满足实例的所有加载操作所需的数量）

4。 完成加载操作

5， 注册写

6。 完成指令

在上述转换之前，之间和之后，任何数量的 伪代码内部步骤 过渡可能会出现。此外， 取指令 在下一个程序位置取指令的过渡将一直有效，直到被采用为止。

这样就结束了对运营模型的非正式描述。以下各节描述了正式的运营模型。

B.3.1指令内伪代码执行

每个指令实例的指令内语义都表示为状态机，实际上是在运行指令伪代码。给定伪代码执行状态，它将计算下一个状态。大多数状态都标识伪代码所请求的未决存储器或寄存器操作，而存储器模型必须执行该操作。状态是（这是一个标记的联合；小写的标记）：

加载内存 *种类，地址，大小，负载继续）* -- 内存加载操作
早期sc失败（ *res延续）* -- 允许 SC 尽早失败
商店ea（ *种类，地址，大小，下一个状态）* -- 内存存储有效地址内存存储值
储存memv（ *mem值，存储连续性）* --
围栏 *善良，下一个状态）* -- 围栏
读reg（ *reg名称，继续阅读）* -- 注册阅读
写reg（ *reg名称，reg值，下一个状态）* -- 注册写
内部（ *下一个状态）* -- 伪代码内部步骤
完成了 -- 伪代码的结尾

这里：

- *记忆值*和 *reg值* 是字节列表；
- *地址* 是XLEN位的整数；
- 用于加载/存储 *类* 识别是否 lr / sc， Acquisition-RCpc / release-RCpc，acquiredRCsc / release-RCsc，acquirement-release-RCsc;
- 围栏 *类* 标识它是正常的还是TSO，以及（对于正常的隔离栅）前驱和后继订购位；

- *注册表名称* 标识一个寄存器及其片（起始和结束位索引）；和
- 后续内容描述了指令实例将如何针对周围内存模型（ *负载连续* 和

阅读延续 取从存储器加载的值并从先前的寄存器写入中读取， *店铺延续* 需要 *假* 为 SC 失败了， *真正* 在其他所有情况下，以及

res延续 需要 *假* 如果 SC 失败并且 *真正* 除此以外）。

例如，给定加载指令 长x1,0（x2）， *执行通常如下。初始执 行状态将根据给定操作码的伪代码计算得出。可以预期* 读reg（ 2倍 *阅读续集）。馈送寄存器的最新写入值* x2（ *指令语义将在必要时被阻塞，直到寄存器值可用为止），例如* 0x4000， *读取连续返回* 加载内存 普通负载，0x4000，

4， *负载连续）。馈送从内存位置加载的4字节值* 0x4000， *说* 0x42，
加载继续收益 写reg（ x1、0x42， 完成）。 *许多*内部（ *下一状态）状态可能会出现在上述状态之前和之间。*

请注意，写入内存分为两个步骤， 商店ea 和 存储memv： 第一个临时使存储的内存占用量已知，第二个将要存储的值相加。我们确保将它们与伪代码配对（ 商店ea 其次是 存储memv），

但它们之间可能还有其他步骤。

可以看出 商店ea *可能在确定要存储的值之前发生。例如，对于运营模型允许使用的石蕊试纸LB + fence.r.rw + data-po（RVWMO是如此），Hart 1中的第一家商店必须采用* 商店ea *在确定其值之前执行第二步，以便第二个存储区可以看到它的存储空间不重叠，从而可以在不破坏一致性的情况下无序地提交第二个存储区。*

每个指令的伪代码最多执行一个存储或一个加载，但恰好执行一个加载和一个存储的AMO除外。然后，这些存储访问通过hart语义分为结构上的原子单元（请参见 启动内存加载操作 和

启动内存存储区操作足迹 下面）。

非正式地，应该通过可以写入该位的最新指令（按程序顺序）从一个寄存器写入中满足该寄存器读取的每个位（如果没有这样的写入，则应从hart的初始寄存器状态中获得）。因此，了解每个指令实例的寄存器写占用空间至关重要，我们将在创建指令实例时对其进行计算（请参见 取指令 下面）。我们在伪代码中确保每个指令最多对每个寄存器位执行一个寄存器写入，并且确保它不尝试读取刚刚写入的寄存器值。

该模型中的数据流相关性（地址和数据）源于以下事实：每个寄存器读取必须等待适当的寄存器写入执行（如上所述）。

B.3.2指令实例状态

每个指令实例 *一世* 状态包括：

- *程序位置* 从中获取指令的内存地址；
- *指示种类* 识别这是装载，存储，AMO，围栏，分支/跳转还是“简单”指令（其中还包括 *类* 与针对伪代码执行状态所描述的类似）；
- *src 吨egs，* 来源集 *注册表名称* s（包括系统寄存器），由指令的伪代码静态确定；
- *dst 吨egs* 目的地 *注册表名称* s（包括系统寄存器），由指令的伪代码静态确定；
- *伪代码状态（* 或有时简称为“状态”），其中之一（这是带标签的联合；小写的标签）：

  普通 *伊萨州)_* -准备进行伪代码转换
  待处理的内存加载（*_负载连续)* _ -请求内存加载操作
  待处理的Mem商店（*_商店延续)* _ -请求内存存储操作

- *reg阅读，* 寄存器读取实例执行的操作，包括每个实例读取的寄存器写切片；
- *reg写道，* 寄存器写入实例已执行；
- *内存加载，* 一组内存加载操作，以及每个内存加载操作尚未满足的片（尚未满足的字节索引），并且对于满足条件的片，还有存储片（每个存储片由内存存储操作和子集组成）满足它的字节索引）。
- *记忆商店* 一组存储器存储操作，每个标志都有一个标志，指示该标志是否已传播（传递到共享存储器）。
- 记录实例是否已提交，完成等的信息

每个内存加载操作都包括一个内存占用量（地址和大小）。每个内存存储操作都包括一个内存占用量以及一个值（如果有）。

具有非空的装入指令实例 *内存加载，* 满足所有负载操作（即不存在不满意的负载切片）的条件是 *完全满意。*

非正式地讲，一个指令实例具有 *完全确定的数据* 如果负载（和 sc） 馈送其源寄存器的指令已完成。同样，据说有一个 *完全确定的内存占用量* 如果负载（和 sc） 输入其存储器操作地址寄存器的指令已完成。形式上，我们首先定义 *完全确定的寄存器写：* 寄存器写

w 从 *reg写* 指令实例 *一世* 据说是 *完全确定* 如果满足以下条件之一：

1。 *一世* 完成; 要么

2.写的值 *w* 不受以下存储操作的影响 *一世* 取得了（即价值
从内存加载或 sc）， 并且，对于每个寄存器，读取 *一世* 已经造成，影响 *w，* 从哪个寄存器写 *一世* 读取已完全确定（或 *一世* 从初始寄存器状态读取）。

现在，一个指令实例 *一世* 据说有 *完全确定的数据* 如果读取每个寄存器 *[R* 从
reg阅读， 寄存器写道 *[R* 读取完全确定。指令实例 *一世* 据说有一个 *完全确定的内存占用量* 如果读取每个寄存器 *[R* 从 *reg阅读* 进入 *一世* 的内存操作地址，寄存器中写道 *[R* 读取完全确定。

的 rmem *在每次执行写操作时，该工具都会为该寄存器的每条记录记录该指令已从其他指令读取的寄存器写集。通过精心安排该工具涵盖的指令的伪代码，我们能够做到这一点，从而使这正是写操作所依赖的一组寄存器写操作。*

B.3.3哈特状态

单个牡鹿的模型状态包括：

- *哈特编号* 牡鹿的唯一标识符；
- *初始寄存器状态* 每个寄存器的初始寄存器值；
- *初始提取地址，*初始指令获取地址；
- *指令树* 按程序顺序获取（但不丢弃）的指令实例的树。

B.3.4共享内存状态

共享内存的模型状态包括内存存储操作的列表，按它们传播到共享内存的顺序。

当存储操作传播到共享内存时，只需将其添加到列表的末尾即可。当从内存满足装入操作时，对于装入操作的每个字节，将返回最新的对应存储片。

在大多数情况下，将共享内存视为一个数组较为简单，即从内存位置到内存存储操作切片的映射，其中每个内存位置都映射到最新的内存存储操作的一个字节切片该位置。但是，此抽象不够详细，无法正确处理 SC *指令。RVWMO 原子公理 允许商店操作与* SC *干预商店的运营* SC *和配对的商店操作* r *阅读。为了允许此类存储操作进行干预，并禁止其他操作，必须扩展数组抽象以记录更多信息。在这里，我们使用一个列表，因为它非常简单，但是更有效和可扩展的实现可能应该使用更好的列表。*

B.3.5过渡

下面的每个段落都描述了一种系统转换。该描述从当前系统状态的条件开始。仅当满足条件时，才能在当前状态下进行转换。该条件之后是在进行转换时应用于该状态的操作，以便生成新的系统状态。

取指令 指令实例的可能的程序顺序继承者 *一世* 可以从地址获取 *位置* 如果：

1. 尚未被获取，即，没有一个直接继承者 *一世* 在牡鹿的
   指令树 来自 *loc;* 和
2. 如果 *一世* 的伪代码已经将地址写入了 *个人电脑* 然后 *位置* 必须是那个地址，否则
   位置 是：
   - 对于条件分支，是后继地址或分支目标地址；
   - 用于（直接）跳转和链接说明（ 贾尔 ）， 目标地址；
   - 用于间接跳转指令（ 贾尔 ）， 任何地址；和
   - 对于其他指令， *一世。 程序位置+* 4。

行动：构造一个新初始化的指令实例 *一世*，用于程序存储器中的指令 *Loc，* 有状态 普通 *isa状态 ），* 根据指令伪代码计算得出，包括伪代码中可用的静态信息，例如其 *指令种类，src regs*，和

dst regs 并添加 *一世*，到牡鹿的 *指令树* 作为...的继任者 *一世。*

可能在获取后立即提供下一个获取地址（ loc ）一世 并且模型不需要等待伪代码写入 pc; *这允许乱序执行，并通过条件分支和跳转进行推测。对于大多数指令，这些地址可以从指令伪代码轻松获得。唯一的例外是间接跳转指令（* 贾尔 ）， *地址取决于寄存器中保存的值。原则上，数学模型应允许此处推测任意地址。详尽搜索*

在里面 rmem *该工具通过多次运行穷举搜索并为每个间接跳转增加一组可能的下一个提取地址来处理此问题。初始搜索使用空集，因此在间接跳转指令之后直到指令的伪代码写入之前都不会进行提取。* 个人电脑 *然后使用该值获取下一条指令。在开始下一次穷举搜索迭代之前，我们为每个间接跳转（按代码位置分组）收集它写入的值集* 个人电脑 *在上一个搜索迭代中的所有执行中使用，并在下一次获取指令的地址时使用该地址。当未检测到新的提取地址时，该过程终止。*

启动内存加载操作 指令实例 *一世* 处于状态 普通（加载内存（ *种类*，*地址*，*大小*，*负载延续* ））可以始终启动相应的内存加载操作。行动：

1.构造适当的内存加载操作 *mlos*：

- 如果 *地址* 对齐到 *尺寸* 然后 *莫洛斯* 是单个内存加载操作 *尺寸* 来自的字节 *地址;*
- 除此以外， *莫洛斯* 是一套 *尺寸* 从地址开始的存储器加载操作，每个字节一个 *地址* 。。。*地址+大小* - 1。

2.设置 *内存加载* 的 *一世* 至 *mlos;* 和

3.更新状态 *一世* 至 待处理的内存加载（ *负载连续*）。

在部分 14.1 据说未对齐的内存访问可能会以任何粒度分解。在这里，我们将它们分解为一个字节的访问，因为这种粒度包含了所有其他访问。

通过从未传播的存储转发来满足内存加载操作 对于非
AMO加载指令实例 *一世* 处于状态 待处理的内存加载（ *负载连续*），和内存加载操作 *o* 在 *一世。* *内存加载* 如果具有未满足条件的切片，则可以通过按程序顺序在前的存储指令实例从未传播的内存存储操作转发来部分或完全满足内存加载操作 *一世* 如果：

1.所有以前的程序订单 围栏 的说明。 sr 和。 w 设置完成；

2.对于每个先前的程序订单 围栏 指令， *F*， 与。 sr 和。 公关 设置和。 w 不
设置，如果 *F* 还没有完成，那么所有程序顺序之前的加载指令 *F* 完全满意；

3.对于每个先前的程序订单 围栏 指令， *F*， 尚未完成，所有负载
之前程序顺序的指令 *F* 完全满意；

4.如果 *一世* 是一个load-acquire-RCsc，所有先前程序顺序的store-releases-RCsc已完成；

5.如果 *一世* 是加载获取释放，所有程序顺序先前的指令均已完成；

6.完全满足所有未完成的先前程序订单的负载获取指令；和

7.之前所有程序订单的商店获取发布指令均已完成；

让 *msoss* 是所有非传播的非传播内存存储操作片的集合 SC 存储之前程序顺序的指令实例 *一世* 并且已经计算出要存储的值，该值与不满足要求的切片重叠 *mlo*， 并且不会被中间存储操作或中间负载读取的存储操作所取代。最后一个条件要求，对于每个内存存储操作片 *msos* 在 *msoss* 从指示 *一世′*：

- 之间没有存储指令program-order *一世* 和 *一世′*内存存储操作重叠 *msos;* 和
- 之间没有加载指令program-order *一世* 和 *一世′*这是来自其他区域的重叠存储操作切片所满足的。

行动：

1.更新 *一世。 内存加载* 表示 *o* 对...感到满意 *msoss;* 和

2.重新启动由此导致的任何违反连贯性的推测性指令，即针对每条未完成的指令 *一世′*那是...的程序顺序继承者 *一世*， 以及每个内存加载操作 *o′*的 *一世′*对此感到满意 *msoss′*， 如果存在内存存储操作片 *msos′*在 *msoss′*， 和来自不同内存存储操作中的重叠内存存储操作分片 *msoss*， 和 *msos′*不是来自是程序的顺序后继项的指令 *一世*， 重新开始 *一世′*及其 *重新启动依赖项。*

在哪里 *重新启动依赖* 教学 *J* 是：

- 的程序顺序继承者 *J* 对寄存器的写入具有数据流依赖性 *j;*
- 的程序顺序继承者 *J* 具有从中读取存储操作的内存加载操作 *j (* 通过转发 ）；
- 如果 *J* 是负载获取，所有的程序顺序继承者 *j;*
- 如果 *J* 对每个人来说都是负担 栅栏 *F*， 与。 sr 和。 公关 设置和。 w 未设置，即是 *j*， 所有的加载指令，它们是 *F;*
- 如果 *J* 对每个人来说都是负担 fence.tso， *F*， 那是...的程序顺序继承者 *j*， 所有的加载指令，它们是 *F;* 和
- （递归）上述所有指令实例的所有重新启动依赖项。

将内存存储操作转发到内存负载可能仅满足部分负载，而无法满足其他负载的需求。

进行上述转换时不可用的先前程序顺序的存储操作可能会使 msoss 暂时变得不健全（违反连贯性）。该商店将阻止完成负载（请参阅 完成指令 ），并在传播该存储操作时使其重新启动（请参见 传播商店运营 ）。

上面的转换条件的结果是store-release-RCsc存储器存储操作无法转发到load-acquire-RCsc指令： msoss 不包含来自已完成存储的存储器存储操作（因为那些操作必须传播到已存储存储器操作中），并且上述条件要求所有程序顺序先前的store-releases-RCsc在加载为acquire-RCsc时都必须完成。

通过内存满足内存加载操作 对于指令实例 *一世* 非AMO加载指令或AMO指令在“ 满足，提交和传播AMO的操作 过渡，任何内存加载操作 *o* 在 *一世。 内存加载* 那不满意

如果满足以下条件，则可以从内存中满足条件： 通过从未传播的存储转发来满足内存加载操作 很满意。行动：让 *msoss* 是内存中覆盖未满足条件的片的内存存储操作片 *mlo*， 并采取行动 通过从未传播的存储转发来满足内存加载操作 。

注意 通过从未传播的存储转发来满足内存加载操作 可能会使内存加载操作的某些部分不令人满意，必须通过再次执行转换或执行 通过内存满足内存加载操作 。 通过内存满足内存加载操作 另一方面，将始终满足所有未满足的内存加载操作要求。

完成 加载 运作 一个 加载 指令 实例 *一世* 在 州 挂起
加载内存（ *负载连续* ）如果所有内存加载操作都可以完成（不要与完成混淆） *一世。 内存加载* 完全满意（即没有不满意的切片）。行动：更新状态 *一世* 至 普通 *负载连续（内存值）*， 哪里 *记忆值*

由满足的所有内存存储操作片组成 *一世。 内存加载。*

早 SC 失败 一个 SC 指令实例 *一世* 处于状态 普通（早期sc失败（ *res延续* ））总是会失败。行动：更新状态 *一世* 至 普通 *res延续（false）*）。

已配对 SC 一个 SC 指令实例 *一世* 处于状态 普通（早期sc失败（ *res延续* ））可以继续执行（可能成功执行），如果 *一世* 与一个配对 先生 行动：更新状态
一世 至 普通 *res延续（true）*）。

启动内存存储区操作足迹 指令实例 *一世* 处于状态
普通（商店ea（ *种类，地址，大小，下一个状态* ））可以随时宣布其未决的内存存储操作足迹。行动：

1.构造适当的内存存储操作 *msos（* 没有商店价值）：

- 如果 *地址* 对齐到 *尺寸* 然后 *msos* 是单个内存存储操作 *尺寸* 字节到 *地址;*
- 除此以外， *msos* 是一套 *尺寸* 内存存储操作，每个操作的字节大小为一个字节 *地址。。。地址+大小* - 1。

2.设置 *一世。 记忆商店* 至 *msos;* 和

3.更新状态 *一世* 至 普通 *下一个状态）*。

请注意，在进行以上转换后，内存存储操作尚无其值。将这种转换与下面的转换分开的重要性在于，它允许其他程序顺序后继存储指令观察该指令的内存占用，并且如果它们不重叠，则应尽早传播（即数据寄存器值变为可用）。

实例化内存存储操作值 指令实例 *一世* 处于状态
普通（存储memv（*_mem值，存储连续性*））总是可以实例化内存存储操作的值 *一世。 内存商店。* 行动：

1.分裂 *记忆值* 内存存储操作之间 *一世。 记忆库；* 和

2.更新状态 *一世* 至 待处理的Mem商店（ *存储延续*）。

提交商店指示 未提交的指令实例 *一世* 非 SC 存储指令或 SC 在“ 提交和传播存储的操作 SC 过渡，处于状态 待处理的Mem商店（ *商店延续*），可以提交（不要与传播混淆）：

1。 *一世* 具有完全确定的数据；

2.所有程序先前的条件分支和间接跳转指令均已完成；

3.所有以前的程序订单 围栏 的说明。 SW 设置完成；

4.所有以前的程序订单 围栏 指示已完成；

5.完成所有先前程序订单的负载获取指令；

6.完成所有先前程序订单的存储获取释放指令；

7.如果 *一世* 是存储版本，所有先前程序订单的指令均已完成；

8.所有先前程序顺序的存储器访问指令均具有完全确定的存储器占用空间；

9.所有先前程序订单的商店指令，除了 SC 失败了，已经开始等等
非空 *记忆库；* 和

10.所有程序先前的加载指令均已启动，因此非空 *内存加载。*

行动：记录下来 *一世* 承诺。

请注意，如果条件 8 满足条件 9 和 10 也很满意，或者在进行了一些急切的过渡后会很满意。因此，要求他们不能加强模型。通过要求它们，我们保证先前的内存访问指令已进行了足够的转换，以使它们的内存操作可见以用于条件检查。 传播商店运营 ，这是指令将执行的下一个转换，使该条件更简单。

传播商店运营 对于已提交的指令实例 *一世* 处于状态 待处理的Mem商店（ *商店延续*），和未传播的内存存储操作 *so* 在
一世。 记忆商店 so 在以下情况下可以传播：

1.与之前的程序顺序存储指令重叠的所有内存存储操作
so 已经传播了；

2.与之前的程序顺序装入指令的所有内存装入操作都与
so 已经满足，并且（加载指令）是 *不可重启（* 参见下面的定义）；和

3.转发满足的所有内存加载操作 *so* 完全满意。

未完成的指令实例 *ʃ* 是 *不可重启* 如果：

1.不存在存储指令 *s* 和未传播的内存存储操作 *so*
的 *s* 以便应用“ 传播商店运营 过渡到 *so* 将导致重新启动 *j;* 和

2.不存在未完成的加载指令 *升* 和内存加载操作 *o* 的
升 以便应用“ 通过从未传播的存储转发来满足内存加载操作 ” /“ 通过内存满足内存加载操作 ”过渡（即使 *o*

已经满足） *o* 将导致重新启动 *j*。

行动：

1.使用以下命令更新共享内存状态 *mso;*

2.更新 *一世。 记忆商店* 表示 *so* 被传播；和

3.重新启动由此导致的任何违反连贯性的推测性指令，即针对每条未完成的指令 *一世′*程序后顺序 *一世* 以及每个内存加载操作

*o′*的 *一世′*对此感到满意 *msoss′*，如果存在内存存储操作片 *msos′*
在 *msoss′*与重叠 *so* 而且不是来自 *味so* 和 *msos′*不是来自的程序订单继承人 *一世*，重新开始 *一世′*及其 *重新启动依赖项（* 看到 通过从未传播的存储转发来满足内存加载操作 ）。

提交和传播存储的操作 SC 一个没有承诺的人 SC 指令实例
一世，来自哈特 *H*，处于状态 待处理的Mem商店（ *商店延续）*，与一对 r *一世′*一些商店已经满足了 *msoss*，在以下情况下可以同时提交和传播：

1。 *一世′*完成;

2.已转发到的每个内存存储操作 *一世′*传播

3.条件 提交商店指示 满意

4.条件 传播商店运营 感到满意（请注意， SC 指令只能
进行一次内存存储操作）；和

5.对于每个商店切片 *msos* 从 *msoss*，*msos* 在共享内存中尚未被覆盖，
来自不是来自牡鹿的商店 *H*，自从 *msos* 被传播到内存。

行动：

1.采取行动 提交商店指示；和

2.采取行动 传播商店运营 。

晚了 SC 失败 一个 SC 指令实例 *一世* 处于状态 待处理的Mem商店（ *商店延续* ），
尚未传播其内存存储操作的数据始终会失败。行动：

1.清除 *一世。 记忆库；* 和

2.更新状态 *一世* 至 普通 *存储连续性（ false ））。*

为了提高效率， rmem *该工具仅在无法使用 提交和传播存储的操作* SC *过渡。这不会影响允许的最终状态集，但是如果进行交互探索，* SC *应该失败一个应该使用 早* SC *失败*

过渡，而不是等待此过渡。

完成 商店 运作 商店指令实例 *一世* 处于状态 挂起
ing mem商店（ *商店延续* ），为其所有内存存储操作 *一世。 记忆商店*
已经传播，可以随时完成（不要与完成混淆）。行动：更新状态 *一世* 至 普通 *存储连续性（ true ））。*

满足，提交和传播AMO的操作 AMO指令实例 *一世* 处于状态 待处理的内存加载（ *负载连续* ）如果可以执行以下过渡序列而无中间过渡，则可以执行其内存访问：

1。 通过内存满足内存加载操作

2。 完成加载操作

3。 伪代码内部步骤 （零次或多次）

4。 实例化内存存储操作值

5， 提交商店指示

6。 传播商店运营

7 完成商店运营

另外， 完成指令 ，除了不需要 *一世* 处于状态 普通（完成）， 在这些过渡之后成立。行动：执行上述转换顺序（不包括 完成指令 ），一个接一个，没有中间过渡。

请注意，以前程序订单的存储不能转发到AMO的负载。这仅仅是因为上面的转换序列不包括转发转换。但是，即使确实包含了该序列，但在尝试执行 传播商店运营

过渡，因为此过渡要求所有程序顺序先前的存储操作都必须重叠

要传播的内存占用量，转发要求不进行存储操作。

另外，不能将AMO的存储转发到程序顺序后继程序。在进行上述转换之前，AMO的存储操作没有其价值，因此无法转发；在完成以上转换之后，存储操作将传播，因此无法转发。

提交围栏 围栏指令实例 *一世* 处于状态 平原（栅栏（ *种类，下一个状态*））在以下情况下可以提交：

1.如果 *一世* 是一个普通的栅栏，它有。 公关 设置后，所有程序顺序先前的加载指令都是
完成

2.如果 *一世* 是一个普通的栅栏，它有。 w 设置，所有程序顺序先前的存储指令都是
完成 和

3.如果 *一世* 是一个 fence.tso， 先前所有程序订单的装入和存储指令均已完成。

行动：

1.记录 *一世* 承诺; 和

2.更新状态 *一世* 至 普通 *下一个状态*）。

注册阅读 指令实例 *一世* 处于状态 普通（阅读reg（ *reg名称，请阅读续*））可以做一个寄存器读取 *注册表名称* 如果它需要读取的每个指令实例都已经执行了预期的 *注册表名称* 注册写。

让 *阅读资料* 包括，每 *reg名称*， 由可以写入该位的最新（按程序顺序）指令实例对该位的写操作。如果没有这样的指令，则源是来自的初始寄存器值 *初始寄存器状态。* 让 *reg值* 是从中收集的价值 *阅读资源。* 行动：

1.添加 *注册表名称* 至 *一世。 reg阅读* 与 *阅读资料* 和 *reg值* 和

2.更新状态 *一世* 至 普通 *读取cont（reg值）*）。

注册写 指令实例 *一世* 处于状态 普通（写reg（ *reg名称，reg值，下一个状态*））总是可以做一个 *注册表名称* 注册写。行动：

1.添加 *注册表名称* 至 *一世。 reg写* 与 *部门* 和 *reg值* 和

2.更新状态 *一世* 至 普通 *下一个状态*）。

哪里 *部门* 是所有的一对 *阅读资料* 从 *一世。 reg阅读*， 和一个真正的标志，当 *一世* 是已经完全满足的加载指令实例。

伪代码内部步骤 指令实例 *一世* 处于状态 普通（内部（ *下一个状态* ））始终可以执行该伪代码内部步骤。行动：更新状态 *一世* 至 普通 *下一个状态*)。

完成指令 一个未完成的指令实例 *一世* 处于状态 普通（完成） 可以在以下情况下完成：

1.如果 *一世* 是加载指令：

（a）已完成所有先前程序订单的负载获取指令；（b）所有先前的程序订单 围栏 的说明。 sr 设置完成；

（c）对于每个先前的程序订单 围栏 指令， *F* ， 尚未完成，所有负载
之前程序顺序的指令 *F* 完成; 和

（d）保证通过以下方式的存储器加载操作读取的值 *一世* 不会造成
一致性违规，即针对任何先前程序顺序的指令实例 *一世′*，让 *cfp*
是从存储指令program-order-between传播的内存存储操作的组合占用空间 *一世* 和 *一世′*，和 *固定内存存储操作* 被转发给
一世 来自商店的指令program-order-between *一世* 和 *一世′*包含 *一世′*，然后让 *cfp* 作为...的补充 *cfp* 在的内存占用量中
一世。 如果 *cfp* 不为空：

一世。 *一世′*具有完全确定的内存占用量；

ii。 *一世′*没有与之重叠的无用的内存存储操作 *cfp;* 和

iii。如果 *一世′*是负载，其内存占用与 *cfp* ， *然后所有的记忆*
的加载操作 *一世′*与...重叠 *cfp* 满意并且 *一世′*是 *不可重启（* 看到 传播商店运营 如何确定指令是否不可重启的转换）
。

在此，如果存储指令具有完全确定的数据，则将存储器存储操作称为固定操作。

2。 *一世* 具有完全确定的数据；和

3.如果 *一世* 不是栅栏，所有程序顺序先前的条件分支和间接跳转指令
完成。

行动：

1.如果 *一世* 是有条件的分支指令或间接跳转指令，舍弃所有未执行的路径
即删除所有分支/跳转无法访问的指令实例 *指令树；* 和

2.记录完成的指令，即设置 *完了* 至 *真正。*

B.3.6局限性

- 该模型涵盖了用户级别的RV64I和RV64A。特别是，它不支持未对齐的原子扩展名“ Zam”或总商店订购扩展名“ Ztso”。它应该是

使模型适应RV32I / A以及G，Q和C扩展很简单，但是我们从未尝试过。这将主要涉及为指令编写Sail代码，而对并发模型的更改最少（如果有的话）。

- 该模型仅涵盖普通内存访问（它不处理I / O访问）。
- 该模型不涵盖与TLB相关的影响。
- 该模型假定指令存储器是固定的。特别是 取指令
  过渡不会生成内存加载操作，并且共享内存不参与过渡。取而代之的是，该模型依赖于外部oracle，该Oracle在给定内存位置时会提供操作码。
- 该模型不涵盖异常，陷阱和中断。

参考书目

[1] RISC-V汇编程序员手册。 https://github.com/riscv/riscv-asm-manual 。

[2] RISC-V ELF psABI规范。 https://github.com/riscv/riscv-elf-psabi-doc/ 。

[3] 32位微处理器的IEEE标准。IEEE标准 1754-1994、1994。

[4] GM Amdahl，GA Blaauw和Jr. FP Brooks。IBM System / 360的体系结构。
IBM研发杂志，1964年第8（2）号。

[5] Werner Buchholz，编辑。 *规划计算机系统：Project Stretch。* 麦格劳-希尔出版社
公司，1962年。

[6] Kourosh Gharachorloo，Daniel Lenoski，James Laudon，Phillip Gibbons，Anoop Gupta和
约翰·轩尼诗 可伸缩共享内存多处理器中的内存一致性和事件排序。在 *在第17届年度国际计算机体系结构研讨会上*，1990年，第
15-26页。

[7]蒂莫西·H·海尔和詹姆斯·E·史密斯。选择性双路径执行。Uni-技术报告
威斯康星州的大学-麦迪逊分校，1996年11月。

[8] ANSI / IEEE Std 754-2008，浮点运算的IEEE标准，2008年。

[9]马诺利斯·卡特尼斯（GH Katevenis），小罗伯特·舍本（Robert W. Sherburne），大卫·帕特森（David A. Patterson）和卡洛·S·奎因（Carlo H.Séequin）。
RISC II微体系结构。在 *VLSI 83会议论文集*，1983年8月。

[10] Hyesoon Kim，Onur Mutlu，Jared Stark和Yale N. Patt。希望分支：结合条件
自适应谓词执行的传统分支和谓词。在 *第38届年度IEEE / ACM国际微体系结构研讨会论文集*，MICRO 38，第43-54页，

2005。

[11] A. Klauser，T。Austin，D。Grunwald和B. Calder。动态吊床预测非
谓词指令集体系结构。在 *1998年国际并行架构和编译技术会议论文集*，PACT '98，美国华盛顿特区，

1998。

[12] David D. Lee，Shing I. Kong，Mark D. Hill，George S. Taylor，David A. Hodges和RandyH。
卡兹（Katz）和大卫·A·帕特森（David A.Patterson）。用于多处理器工作站的VLSI芯片组-第一部分：具有协处理器接口并支持符
号处理的RISC微处理器。 *IEEE JSSC*，1989年12月，第24（6）：1688-1698页。

[13] OpenCores。OpenRISC 1000体系结构手册，体系结构版本1.0,2012年12月。

[14] Heidi Pan，Benjamin Hindman和Krste Asanovi´ć。轻柔：实现高效的合成
并行库。在 *第一届USENIX并行研究热点研讨会论文集（HotPar '09）*，加利福尼亚伯克利，2009年3月。

[15] Heidi Pan，Benjamin Hindman和Krste Asanovi´ć。有效地编写并行软件
和Lithe。在 *第三十一届编程语言设计与实现会议*，
加拿大多伦多，2010年6月。

[16] David A. Patterson和Carlo H. S´equin。RISC I：精简指令集的VLSI计算机。
在 *ISCA*，第443-458页，1981年。

[17]拉维·拉杰瓦尔（Ravi Rajwar）和詹姆斯·古德曼（James R. Goodman）。投机锁选择：实现高度并发的多重
tithreaded执行。在 *第34届ACM / IEEE国际微体系结构国际研讨会论文集*，MICRO 34，第294-305页。IEEE计算机协会，2001
年。

[18] Balaram Sinharoy，R。Kalla，WJ Starke，HQ Le，R。Cargnoni，JA Van Norstrand，
BJ Ronchetti，J。Stuecheli，J.Leenstra，GL Guthrie，DQ Nguyen，B.Blaner，CF Marino，E.Retter和P.Williams。IBM POW
ER7多核服务器处理器。 *IBM研究与发展杂志*，55（3）：1-1，2011年。

[19]詹姆斯·桑顿。在控制数据6600中并行运行。 *十月会议录*
1964年27-29日，秋季联合计算机会议，第二部分：超高速计算机系统，
AFIPS '64（秋季，第二部分），1965年，第33-40页。

[20] Marc Tremblay，Jeffrey Chan，Shailender Chaudhry，AndrewW。Conigliaro和成上
se MAJC体系结构：并行性和可伸缩性的综合。 *IEEE Micro*，20（6）：12–
2000年5月25日。

[21]曾俊杰和阿萨诺维奇·K。节能寄存器访问。在 *进程 第十三届研讨会*
集成电路与系统设计 第377-384页，巴西马瑙斯，2000年9月。

[22] David Ungar，Ricki Blau，Peter Foley，Dain Samples和David Patterson。建筑
SOAR：基于RISC的Smalltalk。在 *ISCA*，第188-197页，密歇根州安阿伯，1984年。

[23]安德鲁·沃特曼。RISC-V Com-提高能源效率并减小代码大小
按下。硕士论文，加州大学伯克利分校，2011年。

[24]安德鲁·沃特曼。 *RISC-V指令集体系结构的设计。* 大学博士学位论文
加州伯克利分校，2016年。

[25]安德鲁·沃特曼（Andrew Waterman），李允素（Yunsup Lee），大卫·A·帕特森（David A. Patterson）和克斯特·阿萨诺维奇（Krste Asanovic）。RISC-V
指令集手册，第I卷：基本用户级ISA。技术报告UCB / EECS-2011-62，加利福尼亚大学伯克利分校EECS系，2011年5月。

[26]安德鲁·沃特曼（Andrew Waterman），李允素（Yunsup Lee），大卫·A·帕特森（David A. Patterson）和克尔斯特·阿萨诺维奇（Krste Asanovic）。RISC-V输入
手册集，第I卷：基本用户级别ISA 2.0版。加州大学伯克利分校EECS系UCB / EECS2014-54技术报告，2014年5月。