The RISC-V Instruction Set Manual

Version Convert pre, 10/2021: Pre-release version

Table of Contents

Preamble

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

Preface

This document describes the RISC-V unprivileged architecture.

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

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

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*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**
Zihintpause	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Zfh	*0.1*	*Draft*
Zfhmin	*0.1*	*Draft*
Zfinx	*0.1*	*Frozen*
Zdinx	*1.0*	*Frozen*
Zhinx	*1.0*	*Frozen*

Base	Version	Status
Zhinxmin	*1.0*	*Frozen*
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.
- Defined PAUSE hint instruction.

Preface to Document Version 20190608-Base-Ratified

This document describes the RISC-V unprivileged architecture.

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

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

Base	Version	Status
RVWMO	2.0	**Ratified**
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
Zifencei	**2.0**	**Ratified**
Zicsr	**2.0**	**Ratified**
M	**2.0**	**Ratified**
A	*2.0*	Frozen
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Ztso	*0.1*	*Frozen*
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
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 *non-conforming*.
- Removed text implying operation under alternate endianness, as alternate-endianness operation has not yet been defined for RISC-V.
- Changed description of misaligned load and store behavior. The specification now allows visible misaligned address traps in execution environment interfaces, rather than just mandating invisible handling of misaligned loads and stores in user mode. Also, now allows access-fault exceptions to be reported for misaligned accesses (including atomics) that should not be emulated.
- Moved FENCE.I out of the mandatory base and into a separate extension, with Zifencei ISA name. FENCE.I was removed from the Linux user ABI and is problematic in implementations with large incoherent instruction and data caches. However, it remains the only standard instruction-fetch coherence mechanism.
- Removed prohibitions on using RV32E with other extensions.
- Removed platform-specific mandates that certain encodings produce illegal instruction exceptions in RV32E and RV64I chapters.
- Counter/timer instructions are now not considered part of the mandatory base ISA, and so CSR instructions were moved into separate chapter and marked as version 2.0, with the unprivileged counters moved into another separate chapter. The counters are not ready for ratification as there are outstanding issues, including counter inaccuracies.
- A CSR-access ordering model has been added.
- Explicitly defined the 16-bit half-precision floating-point format for floating-point instructions in the 2-bit *fmt field*.
- Defined the signed-zero behavior of FMIN.*fmt* and FMAX.*fmt*, and changed their behavior on signaling-NaN inputs to conform to the minimumNumber and maximumNumber operations in the proposed IEEE 754-201x specification.
- The memory consistency model, RVWMO, has been defined.
- The `Zam` extension, which permits misaligned AMOs and specifies their semantics, has been defined.
- The `Ztso` extension, which enforces a stricter memory consistency model than RVWMO, has been defined.

- Improvements to the description and commentary.
- Defined the term IALIGN as shorthand to describe the instruction-address alignment constraint.
- Removed text of P extension chapter as now superseded by active task group documents.
- Removed text of V extension chapter as now superseded by separate vector extension draft document.

Preface to Document Version 2.2

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

Base	*Version*	*Draft Frozen?*
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	Version	Frozen?
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

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

The major changes in this version of the document include:

- The previous version of this document was released under a Creative Commons Attribution 4.0 International License by the original authors, and this and future versions of this document will be released under the same license.
- Rearranged chapters to put all extensions first in canonical order.
- Improvements to the description and commentary.

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

Preface to Document Version 2.1

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

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

- A revised calling convention.
- Relaxed stack alignment for soft-float calling convention, and description of the RV32E calling convention.
- A revised proposal for the C compressed extension, version 1.9 .

Preface to Version 2.0

This is the second release of the user ISA specification, and we intend the specification of the base user ISA plus general extensions (i.e., IMAFD) to remain fixed for future development. The following changes have been made since Version 1.0 (OpenCores, 2012) of this ISA specification.

- The ISA has been divided into an integer base with several standard extensions.
- The instruction formats have been rearranged to make immediate encoding more efficient.
- The base ISA has been defined to have a little-endian memory system, with big-endian or bi-endian as non-standard variants.
- Load-Reserved/Store-Conditional (LR/SC) instructions have been added in the atomic instruction extension.
- AMOs and LR/SC can support the release consistency model.
- The FENCE instruction provides finer-grain memory and I/O orderings.
- An AMO for fetch-and-XOR (AMOXOR) has been added, and the encoding for AMOSWAP has been changed to make room.
- The AUIPC instruction, which adds a 20-bit upper immediate to the PC, replaces the RDNPC instruction, which only read the current PC value. This results in significant savings for position-independent code.
- The JAL instruction has now moved to the U-Type format with an explicit destination register, and the J instruction has been dropped being replaced by JAL with *rd*=`x0`. This removes the only instruction with an implicit destination register and removes the J-Type instruction format from the base ISA. There is an accompanying reduction in JAL reach, but a significant reduction in base ISA complexity.
- The static hints on the JALR instruction have been dropped. The hints are redundant with the *rd* and *rs1* register specifiers for code compliant with the standard calling convention.
- The JALR instruction now clears the lowest bit of the calculated target address, to simplify hardware and to allow auxiliary information to be stored in function pointers.
- The MFTX.S and MFTX.D instructions have been renamed to FMV.X.S and FMV.X.D, respectively. Similarly, MXTF.S and MXTF.D instructions have been renamed to FMV.S.X and FMV.D.X, respectively.
- The MFFSR and MTFSR instructions have been renamed to FRCSR and FSCSR, respectively. FRRM, FSRM, FRFLAGS, and FSFLAGS instructions have been added to individually access the rounding mode and exception flags subfields of the `fcsr`.
- The FMV.X.S and FMV.X.D instructions now source their operands from *rs1*, instead of *rs2*. This change simplifies datapath design.
- FCLASS.S and FCLASS.D floating-point classify instructions have been added.
- A simpler NaN generation and propagation scheme has been adopted.
- For RV32I, the system performance counters have been extended to 64-bits wide, with separate

read access to the upper and lower 32 bits.

- Canonical NOP and MV encodings have been defined.
- Standard instruction-length encodings have been defined for 48-bit, 64-bit, and >64-bit instructions.
- Description of a 128-bit address space variant, RV128, has been added.
- Major opcodes in the 32-bit base instruction format have been allocated for user-defined custom extensions.
- A typographical error that suggested that stores source their data from *rd* has been corrected to refer to *rs2*.

Chapter 1. Introduction

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

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

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

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

The RISC-V ISA is defined avoiding implementation details as much as possible (although commentary is included on implementation-driven decisions) and should be read as the software-visible interface to a wide variety of implementations rather than as the design of a particular hardware artifact. The RISC-V manual is structured in two volumes. This volume covers the design of the base *unprivileged* instructions, including optional unprivileged ISA extensions. Unprivileged instructions are those that are generally usable in all privilege modes in all privileged architectures,

though behavior might vary depending on privilege mode and privilege architecture. The second volume provides the design of the first ("classic") privileged architecture. The manuals use IEC 80000-13:2008 conventions, with a byte of 8 bits.

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

1.1. RISC-V Hardware Platform Terminology

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

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

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

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

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

1.2. RISC-V Software Execution Environments and Harts

The behavior of a RISC-V program depends on the execution environment in which it runs. A RISC-V execution environment interface (EEI) defines the initial state of the program, the number and type of harts in the environment including the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart (i.e., the ISA is one component of the EEI), and the handling of any interrupts or exceptions raised during execution including environment calls. Examples of EEIs include the Linux application binary interface (ABI), or the RISC-V supervisor binary interface (SBI). The implementation of a RISC-V execution environment can be pure hardware, pure software, or a combination of hardware and software. For example, opcode traps and software emulation can be used to implement functionality

not provided in hardware. Examples of execution environment implementations include:

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

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

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

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

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

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

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

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

1.3. RISC-V ISA Overview

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

Although it is convenient to speak of *the* RISC-V ISA, RISC-V is actually a family of related ISAs, of which there are currently four base ISAs. Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers. There are two primary base integer variants, RV32I and RV64I, described in Chapter 2, *RV32I Base Integer Instruction Set, Version 2.1* and Chapter 6, *RV64I Base Integer Instruction Set, Version 2.1*, 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 5, *RV32E Base Integer Instruction Set, Version 1.9* 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 7, *RV128I Base Integer Instruction Set, Version 1.7* 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 32-bit adds in RV32I and 64-bit adds in RV64I with a different opcode ADDW for 32-bit adds in RV64I. This would also be more consistent with the use of the same LW opcode for 32-bit load in both RV32I and RV64I. The very first versions of RISC-V ISA did have a variant of this alternate design, but the RISC-V design was changed to the current choice in January 2011. Our focus was on supporting 32-bit integers in the 64-bit ISA not on providing compatibility with the 32-bit ISA, and the motivation was to remove the asymmetry that arose from having not all opcodes in RV32I have a *W suffix (e.g., ADDW, but AND not ANDW). In hindsight, this was perhaps not well-justified and a consequence of designing both ISAs at the same time as opposed to adding one later to sit on top of another, and also from a belief we had to fold platform requirements into the ISA spec which would imply that all the RV32I instructions would have been required in RV64I. It is too late to change the encoding now, but this is also of little practical consequence for the reasons stated above.

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

RISC-V has been designed to support extensive customization and specialization. Each base integer ISA can be extended with one or more optional instruction-set extensions. An extension may be categorized as either standard, custom, or non-conforming. For this purpose, we divide each RISC-V instruction-set encoding space (and related encoding spaces such as the CSRs) into three disjoint categories: *standard*, *reserved*, and *custom*. Standard extensions and encodings are defined by 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. [extensions] 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 [naming].

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

Beyond the base integer ISA and 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, *`Zifencei` Instruction-Fetch Fence, Version 2.0*.

The memory accesses (implicit or explicit) made by a hart may appear to occur in a different order as perceived by another hart or by any other agent that can access the same memory. This perceived reordering of memory accesses is always constrained, however, by the applicable memory consistency model. The default memory consistency model for RISC-V is the RISC-V Weak Memory Ordering (RVWMO), defined in Chapter 15, *RVWMO Memory Consistency Model, Version 2.0* and in appendices. Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in Chapter 22, *`Ztso` Standard Extension for Total Store Ordering, v0.1*. 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, *C Standard Extension for Compressed Instructions, Version 2.0* 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.

[instlengthcode] illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to 11. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to 00, 01, or 10.

1.5.1. Expanded Instruction-Length Encoding

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

Standard instruction-set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in [instlengthcode]. 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.

			xxxxxxxxxxxxxxaa	16-bit (aa≠`11`)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxbbb11	32-bit (bbb≠`111`)
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx01111 11	48-bit
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx011111 1	64-bit
	···xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx111111 1	(80+16*nnn)-bit, nnn≠`111`
	···xxxx	xxxxxxxxxxxxxxxx	x111xxxxx111111 1	Reserved for ≥192-bits
Byte Address:	base+4	base+2	base	

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

An implementation of the standard IMAFD ISA need only hold the most-significant 30 bits in

instruction caches (a 6.25% saving). On instruction cache refills, any instructions encountered with either low bit clear should be recoded into illegal 30-bit instructions before storing in the cache to preserve illegal instruction exception behavior.

Perhaps more importantly, by condensing our base ISA into a subset of the 32-bit instruction word, we leave more space available for non-standard and custom extensions. In particular, the base RV32I ISA uses less than 1/8 of the encoding space in the 32-bit instruction word. As described in Chapter [extensions], an implementation that does not require support for the standard compressed instruction extension can map 3 additional non-conforming 30-bit instruction spaces into the 32-bit fixed-width format, while preserving support for standard ≥32-bit instruction-set extensions. Further, if the implementation also does not need instructions >32-bits in length, it can recover a further four major opcodes for non-conforming extensions.

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

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

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

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

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

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

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

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

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

1.6. Exceptions, Traps, and Interrupts

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

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

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

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

Contained Trap

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

Requested Trap

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

Invisible Trap

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

Fatal Trap

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

Characteristics of traps; 1) Termination may be requested. 2) shows the characteristics of each kind of trap.

Characteristics of traps; 1) Termination may be requested. 2)

Imprecise fatal traps might be observable by software.

	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

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 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 .

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

Chapter 2. RV32I Base Integer Instruction Set, Version 2.1

This chapter describes the RV32I base integer instruction set.

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

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

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

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

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

2.1. Programmers' Model for Base Integer ISA

Figure 1, "RISC-V base unprivileged integer register state." 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.

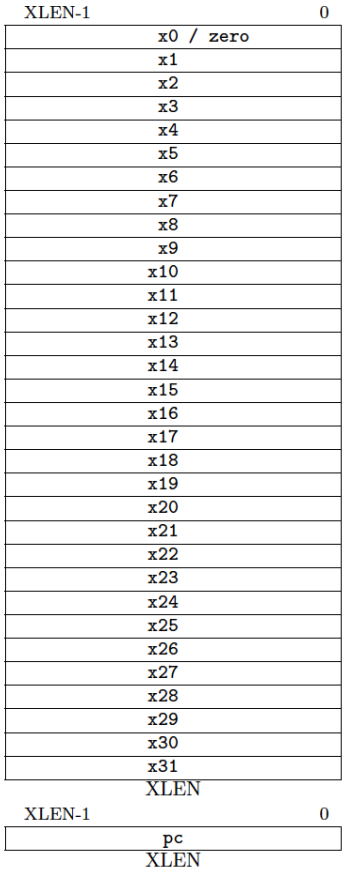


Figure 1. RISC-V base unprivileged integer register state.

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 RV32I. Dynamic register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers. 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 5, *RV32E Base Integer Instruction Set, Version 1.9*).

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, "RISC-V base instruction formats". 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 10, *Zicsr, Control and Status Register (CSR) Instructions, Version 2.0*), 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.

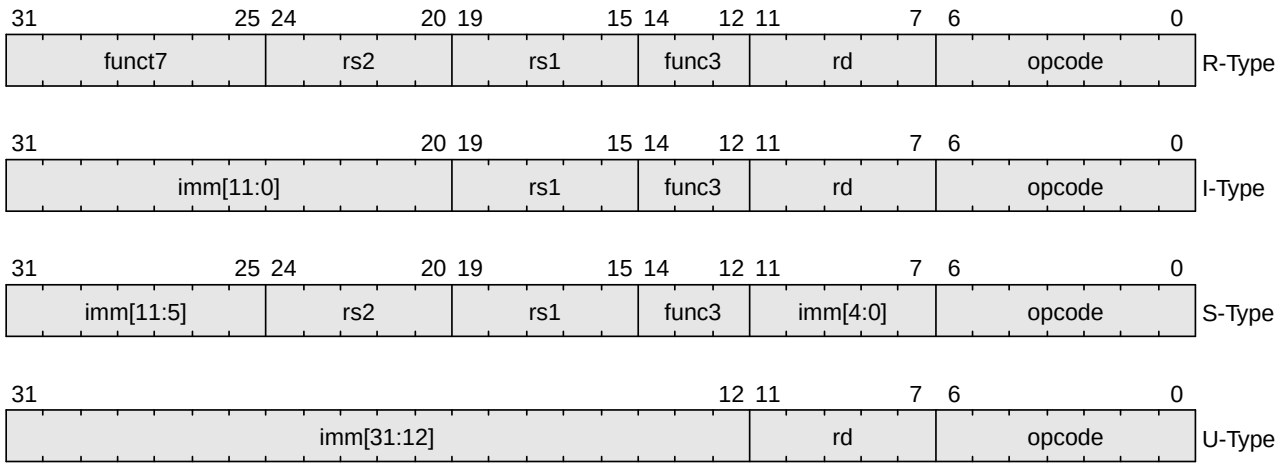


Figure 2. RISC-V base instruction formats

Each immediate subfield in Figure 2, “RISC-V base instruction formats” above is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction’s immediate field as is usually done.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR (Lee et al., 1989)).

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

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

2.3. Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 3, “RISC-V base instruction formats.”.

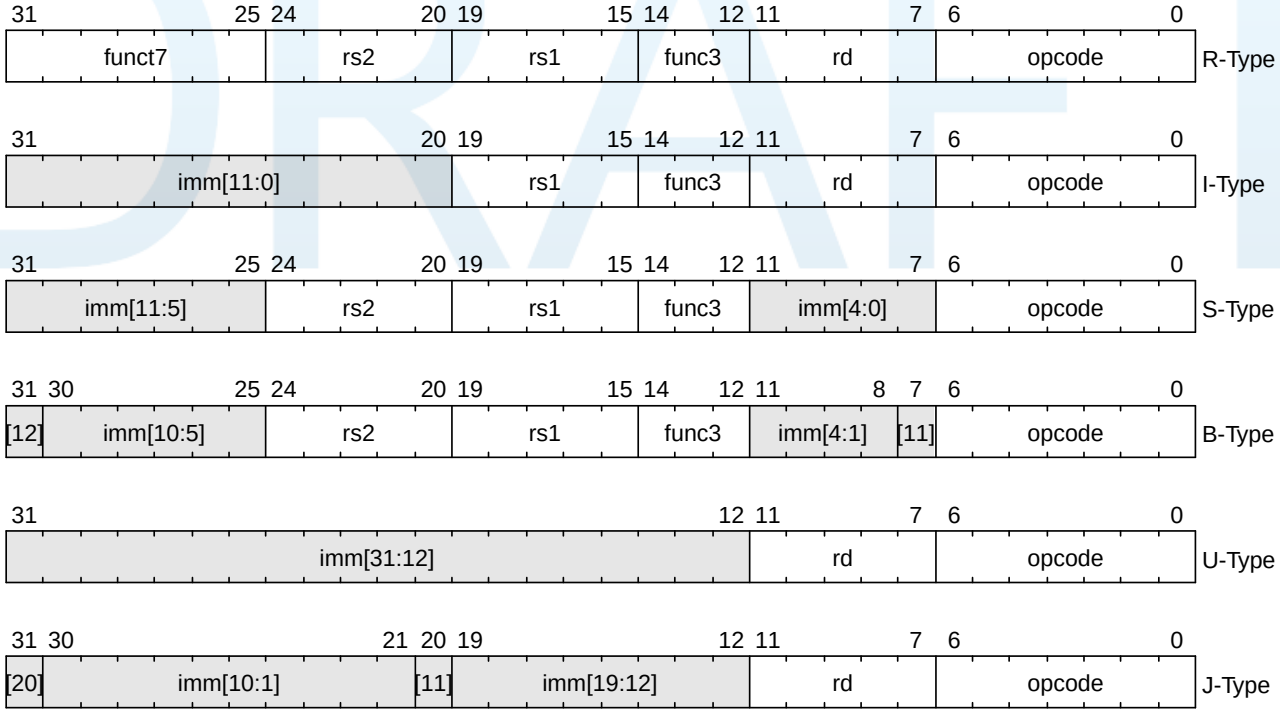


Figure 3. RISC-V base instruction formats.

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 4, "Immediate variants for I, S, B, U, and J" 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.

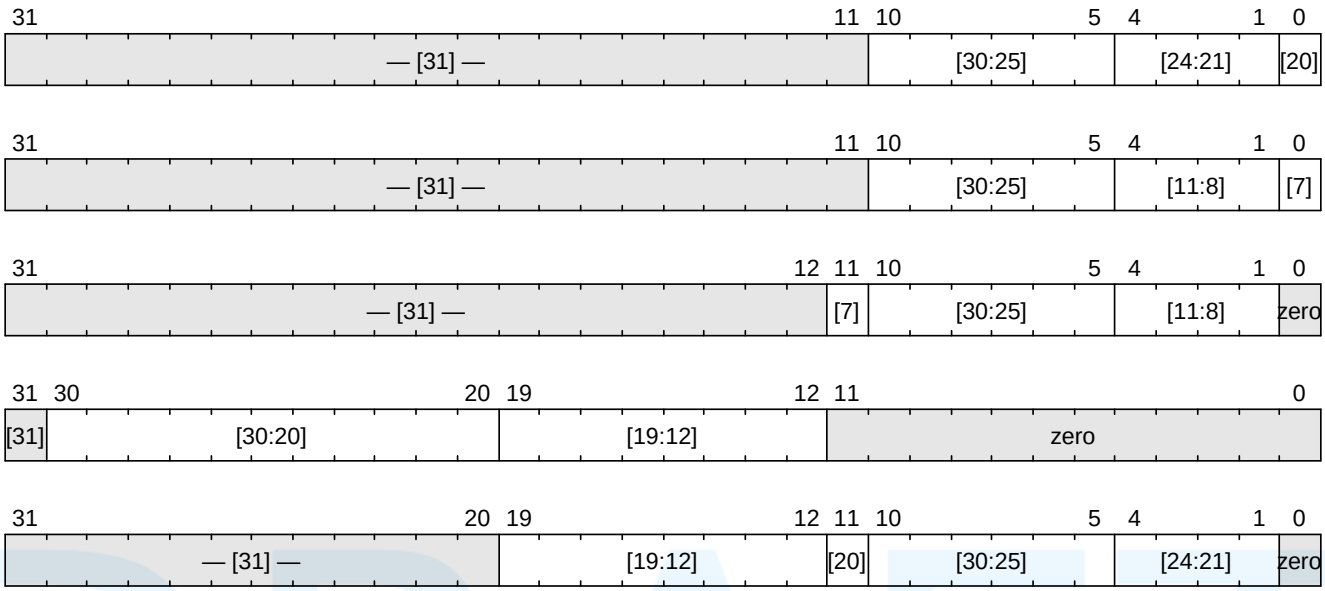


Figure 4. Immediate variants for I, S, B, U, and J

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

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

2.4. Integer Computational Instructions

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

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

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

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

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

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

2.4.1. Integer Register-Immediate Instructions

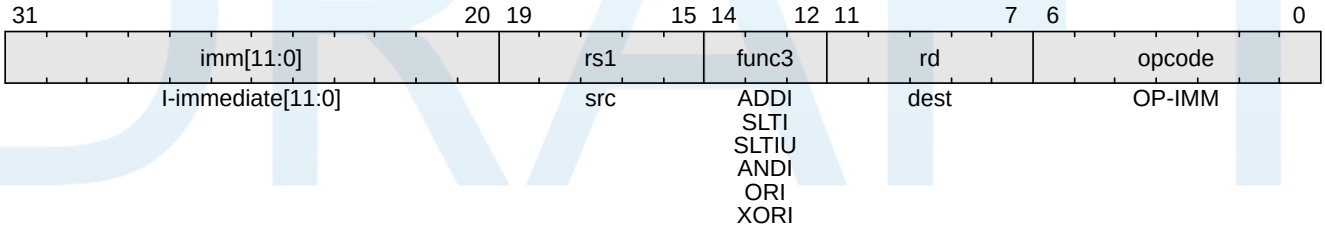


Figure 5. Integer Computational Instructions

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

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

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

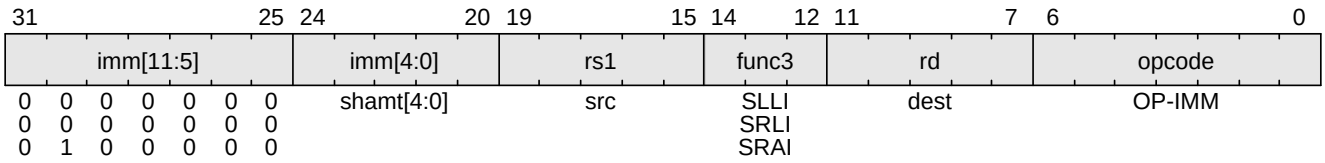


Figure 6. Integer register-immediate, SLLI, SRLI, SRAI

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).

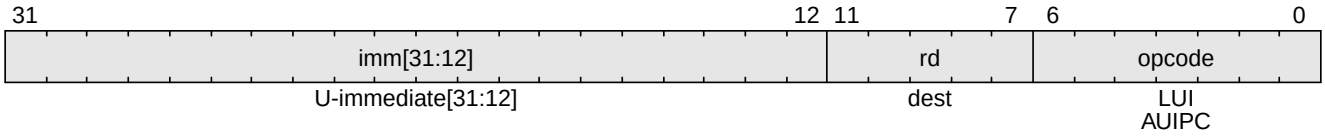


Figure 7. Integer register-immediate, U-immediate

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

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

> *The assembly syntax for `lui` and `auipc` does not represent the lower 12 bits of the U-immediate, which are always zero.*
>
> *The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.*
>
> *The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the local PC (of the instruction following the JAL), it might cause pipeline breaks in simpler microarchitectures or pollute BTB structures in more complex microarchitectures.*

2.4.2. Integer Register-Register Operations

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

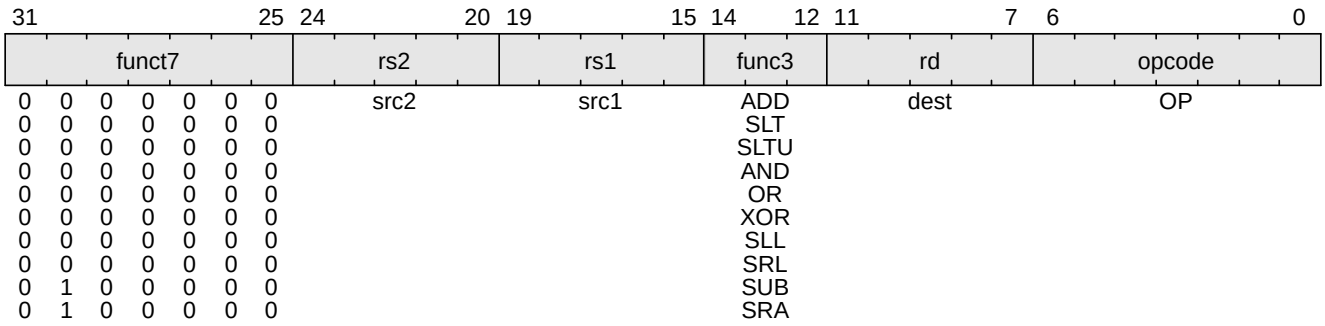


Figure 8. Integer register-register

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

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

2.4.3. NOP Instruction

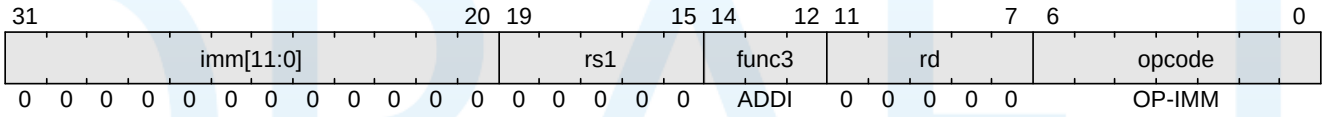


Figure 9. NOP instructions

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 [rv32i-hints]).*
>
> *ADDI was chosen for the NOP encoding as this is most likely to take fewest resources to execute across a range of systems (if not optimized away in decode). In particular, the instruction only reads one register. Also, an ADDI functional unit is more likely to be available in a superscalar design as adds are the most common operation. In particular, address-generation functional units can execute ADDI using the same hardware needed for base+offset address calculations, while register-register ADD or logical/shift operations require additional hardware.*

2.5. Control Transfer Instructions

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

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

2.5.1. Unconditional Jumps

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

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

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

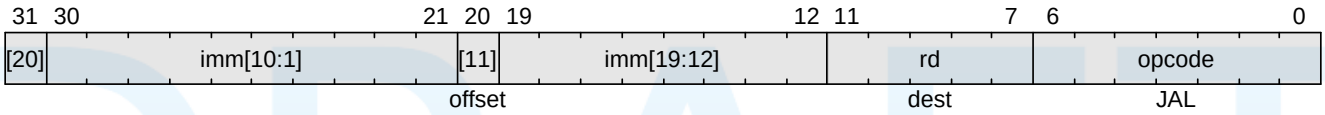


Figure 10. The unconditional-jump instruction, JAL

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.

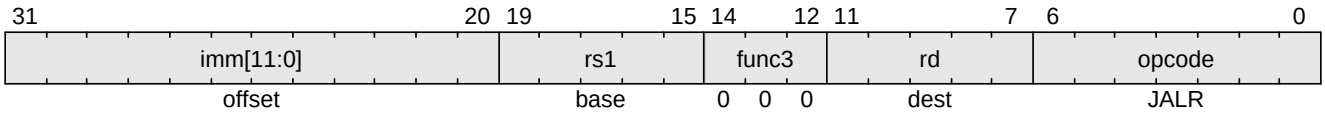


Figure 11. The indirect unconditional-jump instruction, JALR

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

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

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

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

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

Instruction-address-misaligned exceptions are not possible on machines 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 Table 1, "Return-address stack prediction hints encoded in the register operands of a JALR instruction.".

Table 1. Return-address stack prediction hints encoded in the register operands of a JALR instruction.

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

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

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

2.5.2. Conditional Branches

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

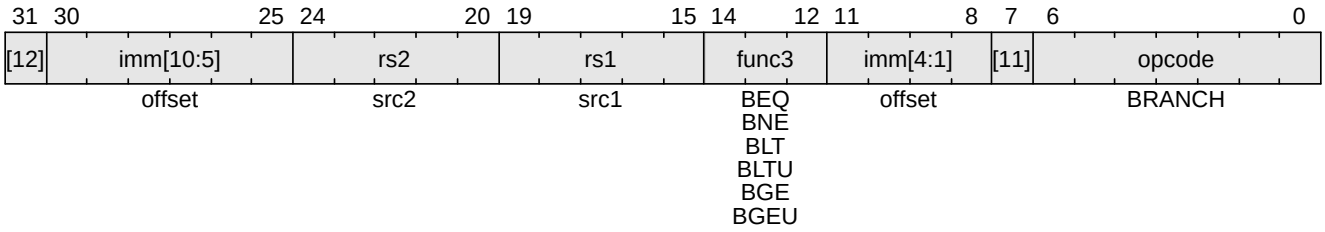


Figure 12. Conditional branches

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

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

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

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

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

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

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

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict (Heil & Smith, 1996), (Klauser et al., 1998), (Kim et al., 2005) and have been implemented in commercial processors (Sinharoy et al., 2011). The simplest techniques just reduce the penalty of recovering from a mispredicted short forward branch by only flushing instructions in the branch shadow instead of the entire fetch pipeline, or by fetching instructions from both sides using wide instruction fetch or idle instruction fetch slots. More complex techniques for out-of-order cores add internal predicates on instructions in the branch shadow, with the internal predicate value written by the branch instruction, allowing the branch and following instructions to be executed speculatively and out-of-order with respect to other code.

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

Instruction-address-misaligned exceptions are not possible on machines that support extensions with

16-bit aligned instructions, such as the compressed instruction-set extension, C.

2.6. Load and Store Instructions

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

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

> *In a system for which endianness is byte-address invariant, the following property holds: if a byte is stored to memory at some address in some endianness, then a byte-sized load from that address in any endianness returns the stored value.*
>
> *In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes.*
>
> *In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.*

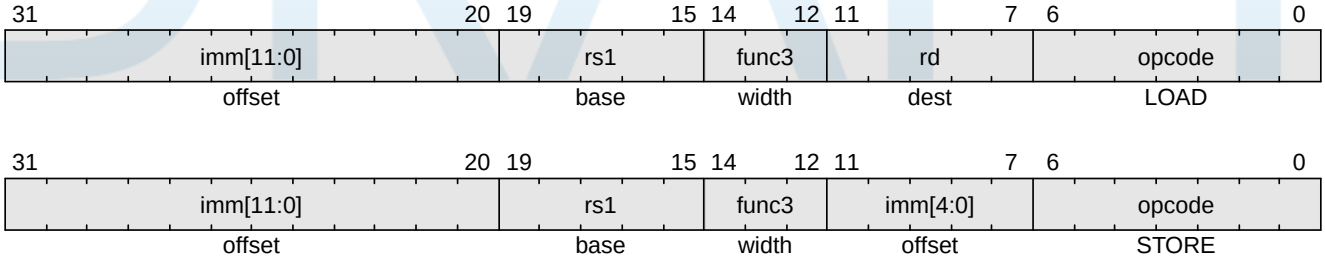


Figure 13. Load and store instructions

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

The LW instruction loads a 32-bit value from memory into *rd*. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in *rd*. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in *rd*. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory.

Regardless of EEI, loads and stores whose effective addresses are naturally aligned shall not raise an address-misaligned exception. Loads and stores whose effective address is not naturally aligned to the

referenced datatype (i.e., the effective address is not divisible by the size of the access in bytes) have behavior dependent on the EEI.

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

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

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

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

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

2.7. Memory Ordering Instructions

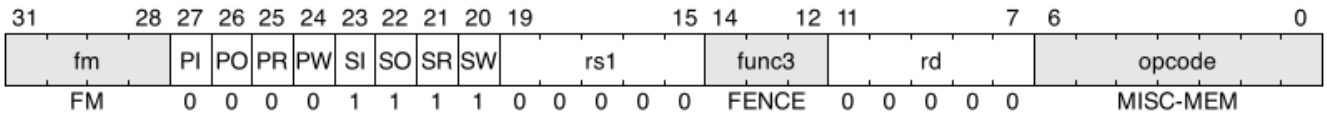


Figure 14. Memory ordering instructions

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

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

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

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

Table 2. Fence mode encoding

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

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

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

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

The unused fields in the FENCE instructions--*rs1* and *rd*--are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields. Likewise, many *fm* and predecessor/successor set settings in Table 2, "Fence mode encoding" 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 10, *Zicsr, Control and Status Register (CSR) Instructions, Version 2.0*, 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.

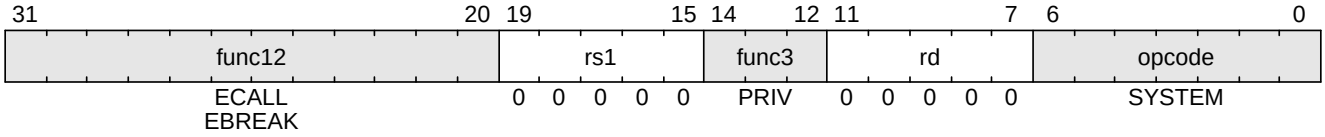


Figure 15. Evironment call and breakpoint instructions

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

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

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

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

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

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

```
slli x0, x0, 0x1f   # Entry NOP
ebreak              # Break to debugger
srai x0, x0, 7      # 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, C Standard Extension for Compressed Instructions, Version 2.0.

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

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

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

2.9. HINT Instructions

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

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

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

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

Table 3, "RV32I HINT instructions." lists all RV32I HINT code points. 91% of the HINT space is reserved for standard HINTs. The remainder of the HINT space is designated for custom HINTs: no standard HINTs will ever be defined in this subspace.

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

Table 3. RV32I HINT instructions.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1* ≠ x0 or *imm*≠0	$2^{17} - 1$	
ANDI	*rd*=x0	2^{17}	
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	*rd*=x0, *rs1* ≠ x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10} - 63$	
FENCE	*rd* ≠ x0, *rs1*=x0, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10} - 63$	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=0, *succ*≠0	15	
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*≠W, *succ*≠0	15	*Reserved for future standard use*
FENCE	*rd*=*rs1*=x0, *fm*=0, *pred*=W, *succ*=0	1	PAUSE
SLTI	*rd*=x0	2^{17}	
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{10}	
SRLI	*rd*=x0	2^{10}	
SRAI	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	*Designated for custom use*
SLTU	*rd*=x0	2^{10}	

Chapter 3. Zifencei Instruction-Fetch Fence, Version 2.0

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

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

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

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

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

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

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

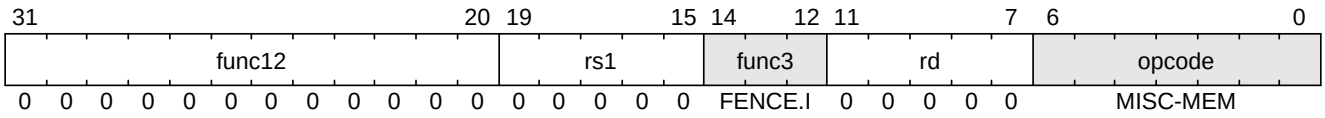


Figure 16. FENCE.I instruction

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. "Zihintpause" Pause Hint, Version 2.0

The PAUSE instruction is a HINT that indicates the current hart's rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero. No architectural state is changed.

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

A future extension might add primitives similar to the x86 MONITOR/MWAIT instructions, which provide a more efficient mechanism to wait on writes to a specific memory location. However, these instructions would not supplant PAUSE. PAUSE is more appropriate when polling for non-memory events, when polling for multiple events, or when software does not know precisely what events it is polling for.

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

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

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

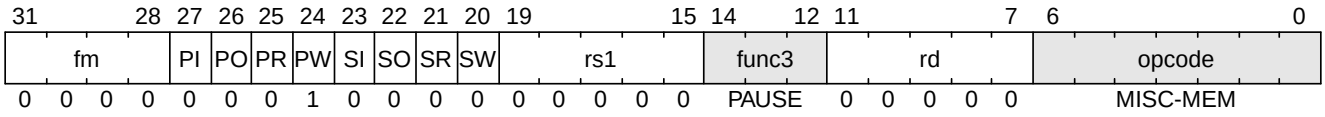


Figure 17. Zihintpause fence instructions

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

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

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

Chapter 5. 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, *RV32I Base Integer Instruction Set, Version 2.1*.

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.

5.1. 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.

5.2. 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 6. 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, *RV32I Base Integer Instruction Set, Version 2.1*. This chapter presents only the differences with RV32I, so should be read in conjunction with the earlier chapter.

6.1. Register State

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

6.2. 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.

6.2.1. Integer Register-Immediate Instructions

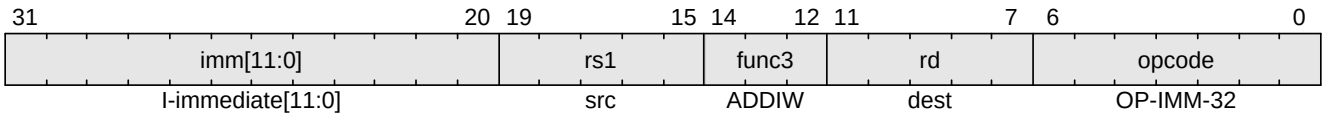
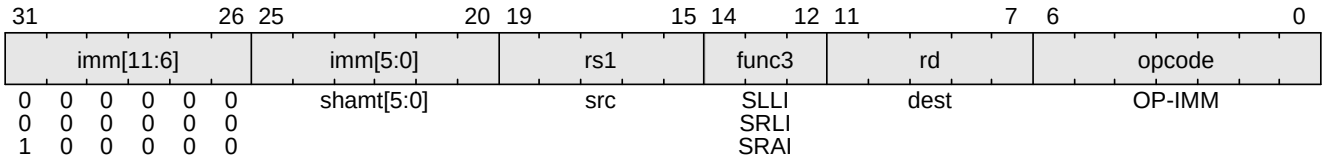


Figure 18. RV64I register-immediate instructions

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



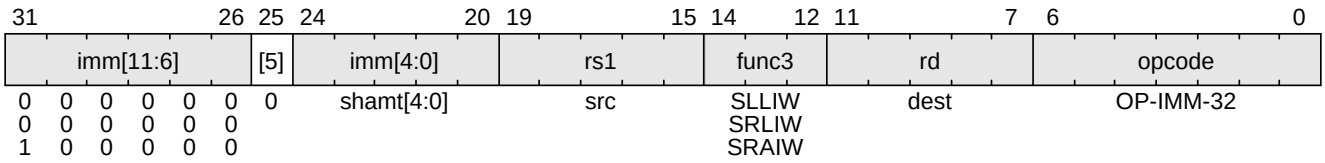


Figure 19. RV64I register-immediate (descr ADDIW) instructions

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

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

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

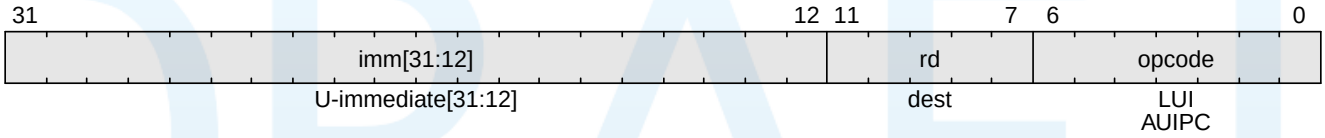


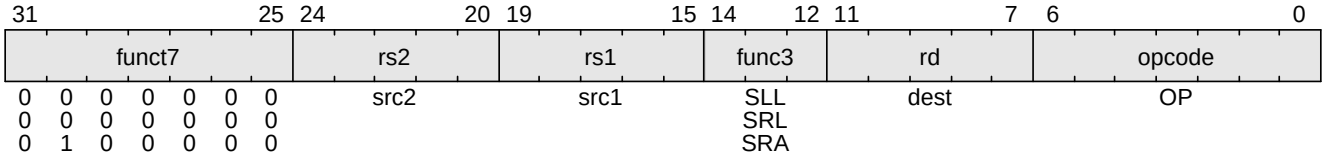
Figure 20. RV64I register-immediate (descr) instructions

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

6.2.2. Integer Register-Register Operations



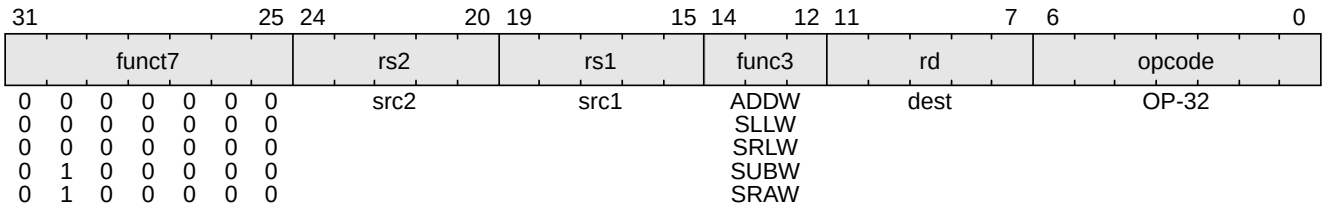


Figure 21. RV64I integer register-register instructions

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]*.

6.3. 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.

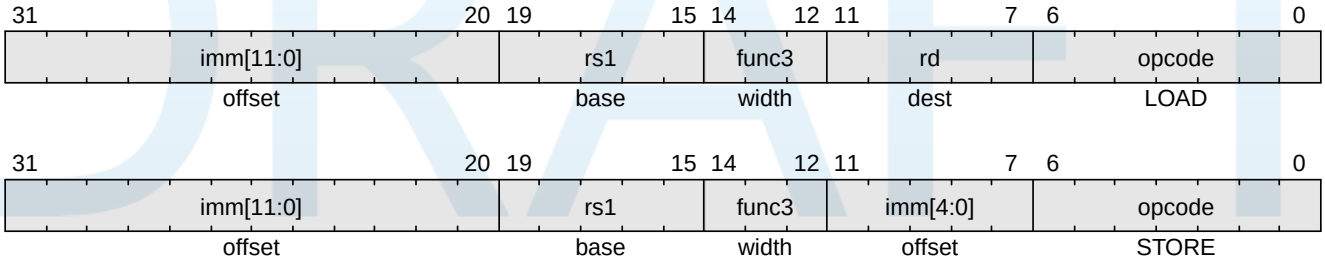


Figure 22. Load and store instructions

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.

6.4. HINT Instructions

All instructions that are microarchitectural HINTs in RV32I (see Chapter 2, *RV32I Base Integer Instruction Set, Version 2.1* are also HINTs in RV64I. The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.

Table 4, "RV64I HINT instructions." lists all RV64I HINT code points. 91% of the HINT space is reserved for standard HINTs, but none are presently defined. The remainder of the HINT space is

designated for custom HINTs; no standard HINTs will ever be defined in this subspace.

Table 4. RV64I HINT instructions.

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	*rd=x0*, *rs1_ ≠ _x0*, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10} - 63$	
	rd_ ≠ _x0, *rs1=x0*, *fm*=0, and either *pred*=0 or *succ*=0	$2^{10} - 63$	
	rd=rs1=x0, *fm*=0, *pred*=0, _succ_ ≠ 0	15	
	pred=0 or *succ*=0, *pred_ ≠ W*, *_succ*=0	15	
	rd=rs1=x0, *fm*=0, *pred*=W, *succ*=0	1	PAUSE

Instruction	Constraints	Code Points	Purpose
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}	

Chapter 7. RV128I Base Integer Instruction Set, Version 1.7

"There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management." Bell and Strecker, ISCA-3, 1976.

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

The primary reason to extend integer register width is to support larger address spaces. It is not clear when a flat address space larger than 64 bits will be required. At the time of writing, the fastest supercomputer in the world as measured by the Top500 benchmark had over 1PB of DRAM, and would require over 50 bits of address space if all the DRAM resided in a single address space. Some warehouse-scale computers already contain even larger quantities of DRAM, and new dense solid-state non-volatile memories and fast interconnect technologies might drive a demand for even larger memory spaces. Exascale systems research is targeting 100PB memory systems, which occupy 57 bits of address space. At historic rates of growth, it is possible that greater than 64 bits of address space might be required before 2030.

History suggests that whenever it becomes clear that more than 64 bits of address space is needed, architects will repeat intensive debates about alternatives to extending the address space, including segmentation, 96-bit address spaces, and software workarounds, until, finally, flat 128-bit address spaces will be adopted as the simplest and best solution.

We have not frozen the RV128 spec at this time, as there might be need to evolve the design based on actual usage of 128-bit address spaces.

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

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

Shifts by an immediate (SLLI/SRLI/SRAI) are now encoded using the low 7 bits of the I-immediate,

and variable shifts (SLL/SRL/SRA) use the low 7 bits of the shift amount source register.

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

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

Chapter 8. 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.

8.1. Multiplication Operations

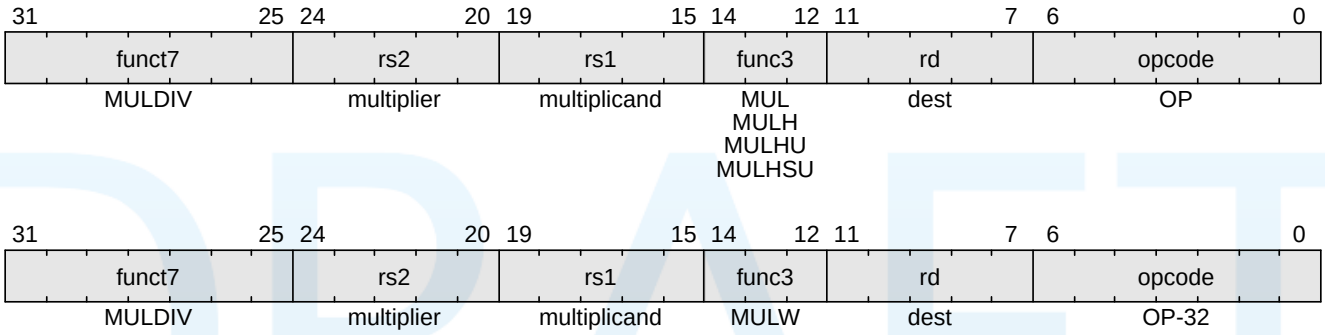


Figure 23. Multiplication operation instructions

MUL performs an XLEN-bit X 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 X XLEN-bit product, for signed X signed, unsigned X unsigned, and *rs1X* unsigned *rs2* multiplication, respectively. If both the high and low bits of the same product are required, then t he 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].

8.2. Division Operations

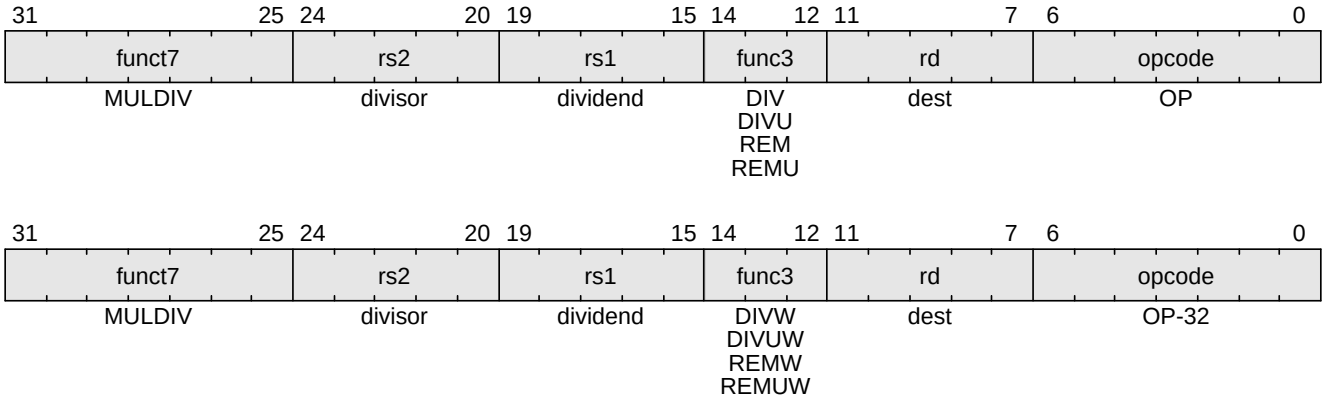


Figure 24. Division operation instructions

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

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

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

The semantics for division by zero and division overflow are summarized in Table 5, “Semantics for division by zero and division overflow.”. The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by −1. The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

Table 5. Semantics for division by zero and division overflow.

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

In Table 5, “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.

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

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

8.3. Zmmul Extension, Version 0.1

The Zmmul extension implements the multiplication subset of the M extension. It adds all of the instructions defined in Section 8.1, "Multiplication Operations", namely: MUL, MULH, MULHU, MULHSU, and (for RV64 only) MULW. The encodings are identical to those of the corresponding M-extension instructions.

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

Chapter 9. A Standard Extension for Atomic Instructions, Version 2.1

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

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

9.1. Specifying Ordering of Atomic Instructions

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

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

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

9.2. Load-Reserved/Store-Conditional Instructions

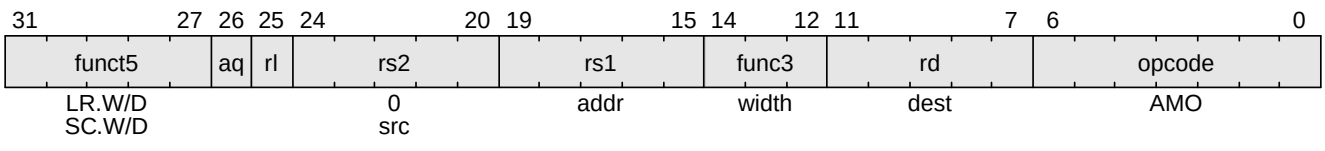


Figure 25. Load-Reserved/Store-Conditional Instructions

Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR.W loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a *reservation set*—a set of bytes that subsumes the bytes in the addressed word. SC.W conditionally writes a word in *rs2* to the address in *rs1*: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the SC.W succeeds, the instruction writes the word in *rs2* to memory, and it writes zero to *rd*. If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to *rd*. Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart. LR.D and SC.D act analogously on doublewords and are only available on RV64. For RV64, LR.W and SC.W sign-extend the value placed in *rd*.

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

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

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

The failure code with value 1 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 15.1, "Definition of the RVWMO Memory Model".

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 15.1, "Definition of the RVWMO Memory Model" 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.

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

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

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 [cas]. If inlined, compare-and-swap functionality need only take four instructions.

9.3. Eventual Success of Store-Conditional Instructions

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

- The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory.
- An LR/SC sequence begins with an LR instruction and ends with an SC instruction. The dynamic code executed between the LR and SC instructions can only contain instructions from the base *I* instruction set, excluding loads, stores, backward jumps, taken backward branches, JALR, FENCE,

and SYSTEM instructions. If the *C* extension is supported, then compressed forms of the aforementioned *I* instructions are also permitted.

- The code to retry a failing LR/SC sequence can contain backwards jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraint as the code between the LR and SC.
- The LR and SC addresses must lie within a memory region with the *LR/SC eventuality* property. The execution environment is responsible for communicating which regions have this property.
- The SC must be to the same effective address and of the same data size as the latest LR executed by the same hart.

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

We restricted the length of LR/SC loops to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and associativity. Similarly, we disallowed other loads and stores within the loops to avoid restrictions on data-cache associativity in simple implementations that track the reservation within a private cache. The restrictions on branches and jumps limit the time that can be spent in the sequence. Floating-point operations and integer multiply/divide were disallowed to simplify the operating system's emulation of these instructions on implementations lacking appropriate hardware support.

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

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

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

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

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

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

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

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

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

9.4. Atomic Memory Operations

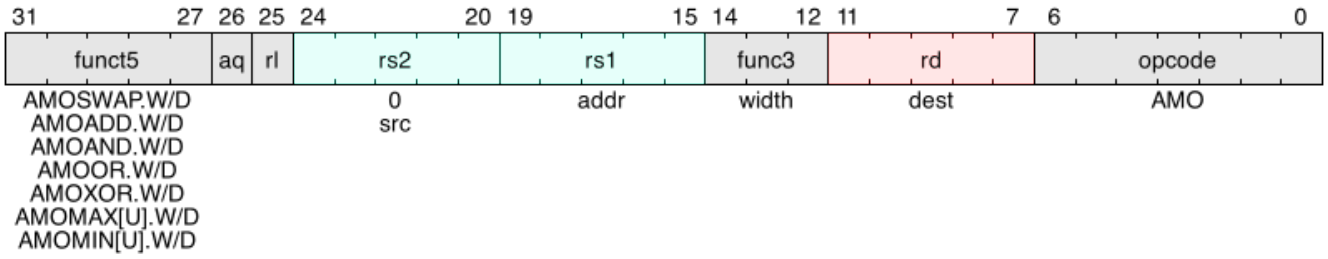


Figure 26. Atomic memory operations

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

For AMOs, the A extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for 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. The *Zam* extension, described in Chapter 21, *Zam Standard Extension for Misaligned Atomics, v0.1*, relaxes this requirement and specifies the semantics of misaligned AMOs.

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

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

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

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

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

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

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

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

The instructions in the *A* extension can also be used to provide sequentially consistent loads and stores. A sequentially consistent load can be implemented as an LR with both *aq* and *rl* set. A sequentially consistent store can be implemented as an AMOSWAP that writes the old value to x0 and has both *aq* and *rl* set.

Chapter 10. Zicsr, Control and Status Register (CSR) Instructions, Version 2.0

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

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

The counters and timers are no longer considered mandatory parts of the standard base ISAs, and so the CSR instructions required to access them have been moved out of Chapter 2, RV32I Base Integer Instruction Set, Version 2.1 into this separate chapter.

10.1. CSR Instructions

CSR

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

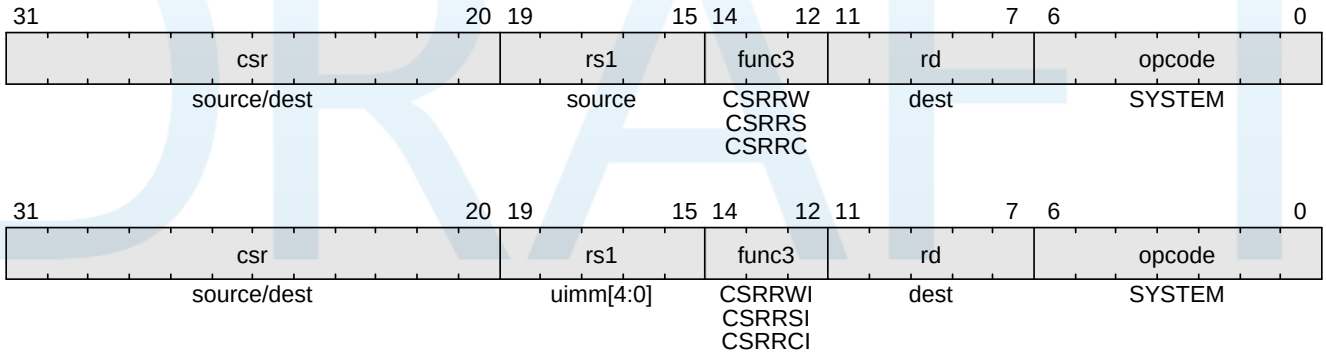


Figure 27. CSR instructions

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

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

treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. 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 6. Conditions determining whether a CSR instruction reads or writes the specified CSR.

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

Table 6, “Conditions determining whether a CSR instruction reads or writes the specified CSR.” 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.

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

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

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

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

10.1.1. CSR Access Ordering

Each RISC-V hart normally observes its own CSR accesses, including its implicit CSR accesses, as performed in program order. In particular, unless specified otherwise, a CSR access is performed after the execution of any prior instructions in program order whose behavior modifies or is modified by the CSR state and before the execution of any subsequent instructions in program order whose behavior modifies or is modified by the CSR state. Furthermore, an explicit CSR read returns the CSR state before the execution of the instruction, while an 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 15, *RVWMO Memory Consistency Model, Version*

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

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

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

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

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

The rules for the reordering of CSR accesses in the global memory order should probably be moved to Chapter 15, RVWMO Memory Consistency Model, Version 2.0 concerning the RVWMO memory consistency model.

Chapter 11. Counters

RISC-V ISAs provide a set of up to 32_X_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.

11.1. Base Counters and Timers

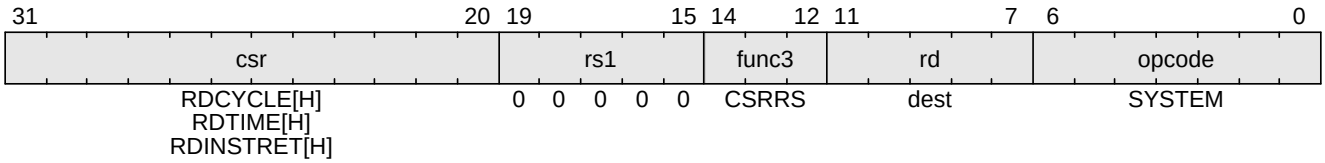


Figure 28. Base counters and timers

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12-bit 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" difficult given some implementation choices (e.g., AMD Bulldozer). Precisely defining what is a "clock cycle" is also difficult given the range of implementations (including software emulations), but the intent is that RDCYCLE is used for performance monitoring along with the other performance counters. In particular, where there is one hart/core, one would expect cycle-count/instructions-retired to measure CPI for a hart.

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

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

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

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

The RDTIME pseudoinstruction reads the low XLEN bits of the **time** CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is 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.

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

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

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.

11.2. 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 platform-specific 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 12. F Standard Extension for Single-Precision Floating-Point, Version 2.2

This chapter describes the standard instruction-set extension for single-precision floating-point, which is named `F` and adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard (*ANSI/IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic*, 2008). The F extension depends on the `Zicsr` extension for control and status register access.

12.1. 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 29, "RISC-V standard F extension single-precision floating-point state". 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.

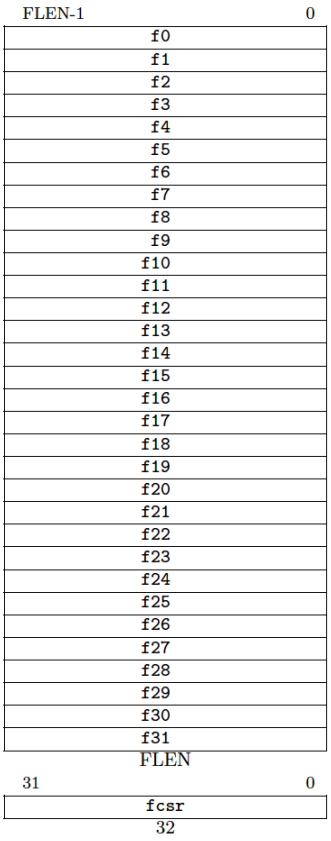


Figure 29. RISC-V standard F extension single-precision floating-point state

12.2. 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 30, "Floating-point control and status register".

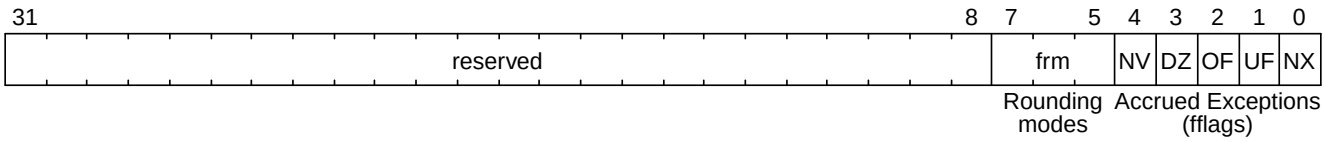


Figure 30. Floating-point control and status register

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. 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 7, "Rounding mode encoding.". A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101–110) and dynamic reserved rounding modes (101–111). Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

Table 7. Rounding mode encoding.

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

The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

The ratified version of the F spec mandated that an illegal instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 8, "Accrued exception flag encoding.". The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 8. Accrued exception flag encoding.

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

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

12.3. 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.

12.4. 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.

12.5. Single-Precision Load and Store Instructions

Floating-point loads and stores use the same base+offset addressing mode as the integer base ISAs, with a base address in register *rs1* and a 12-bit signed byte offset. The FLW instruction loads a single-precision floating-point value from memory into floating-point register *rd*. FSW stores a single-precision value from floating-point register *rs2* to memory.

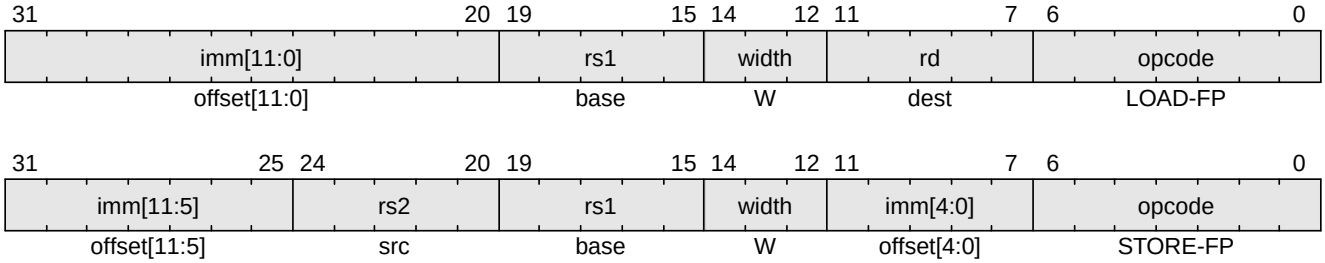


Figure 31. SP load and store

FLW and FSW are only guaranteed to execute atomically if the effective address is naturally aligned.

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

As described in Section 2.6, "Load and Store Instructions", the execution environment defines whether misaligned floating-point loads and stores are handled invisibly or raise a contained or fatal trap.

12.6. Single-Precision Floating-Point Computational Instructions

Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode. FADD.S and FMUL.S perform single-precision floating-point addition and multiplication respectively, between *rs1* and *rs2*. FSUB.S performs the single-precision floating-point subtraction of *rs2* from *rs1*. FDIV.S performs the single-precision floating-point division of *rs1* by *rs2*. FSQRT.S computes the square root of *rs1*. In each case, the result is written to *rd*.

The 2-bit floating-point format field *fmt* is encoded as shown in Table 9, "Format field encoding". It is set to *S* (00) for all instructions in the F extension.

Table 9. Format field encoding

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

All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in Table 7, "Rounding mode encoding.".

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.

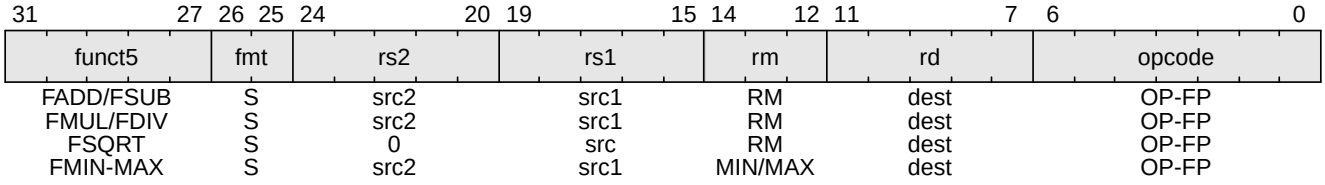


Figure 32. Single-Precision Floating-Point Computational Instructions

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.

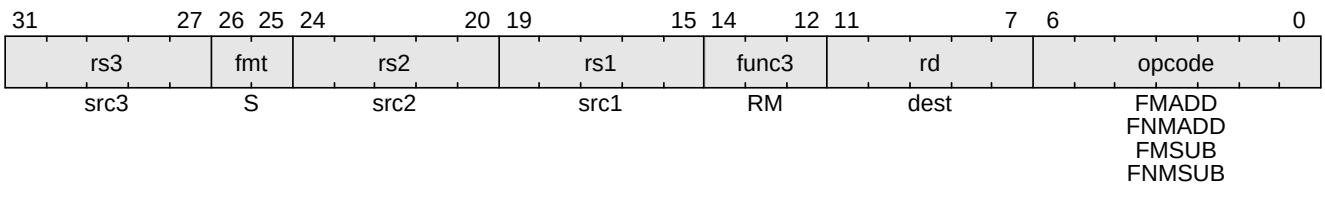


Figure 33. F[N]MADD/F[N]MSUB instructions

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.

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

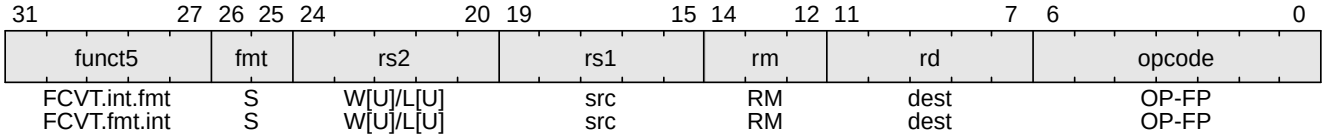
Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.S or FCVT.L.S converts a floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.S.W or FCVT.S.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a floating-point number in floating-point register *rd*. FCVT.WU.S, FCVT.LU.S, FCVT.S.WU, and FCVT.S.LU variants convert to or from unsigned integer values. For XLEN>32, FCVT.W[U].S sign-extends the 32-bit result to the destination register width. FCVT.L[U].S and FCVT.S.L[U] are RV64-only instructions. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. Table 10, "Domains of float-to-integer conversions and behavior for invalid inputs" gives the range of valid inputs for FCVT.*int*.S and the behavior for invalid inputs.

Table 10. Domains of float-to-integer conversions and behavior for invalid inputs

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

All floating-point 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.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.




```
{reg: [
  {bits: 7, name: 'opcode', attr: ['OP-FP'],    type: 8},
  {bits: 5, name: 'rd',     attr: ['dest'],     type: 2},
  {bits: 3, name: 'rm',  attr: ['J[N]/JX'],      type: 8},
  {bits: 5, name: 'rs1',    attr: ['src1'],      type: 4},
  {bits: 5, name: 'rs2',    attr: ['src2'],     type: 8},
  {bits: 2, name: 'fmt',    attr: ['S'],        type: 8},
  {bits: 5, name: 'funct5', attr: ['FSGNJ'],    type: 8},
]}
```

Figure 34. SP float convert and move

Floating-point to floating-point sign-injection instructions, FSGNJ.S, FSGNJN.S, and FSGNJX.S, produce a result that takes all bits except the sign bit from *rs1*. For FSGNJ, the result's sign bit is *rs2*'s sign bit; for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit; and for FSGNJX, the sign bit is the XOR of the sign bits of *rs1* and *rs2*. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs. Note, FSGNJ.S *rx, ry, ry* moves *ry* to *rx* (assembler pseudoinstruction FMV.S *rx, ry*); FSGNJN.S *rx, ry, ry* moves the negation of *ry* to *rx* (assembler pseudoinstruction FNEG.S *rx, ry*); and FSGNJX.S *rx, ry, ry* moves the absolute value of *ry* to *rx* (assembler pseudoinstruction FABS.S *rx, ry*).

The sign-injection instructions provide floating-point MV, ABS, and NEG, as well as supporting a few other operations, including the IEEE copySign operation and sign manipulation in transcendental math function libraries. Although MV, ABS, and NEG only need a single register operand, whereas FSGNJ instructions need two, it is unlikely most microarchitectures would add optimizations to benefit from the reduced number of register reads for these relatively infrequent instructions. Even in this case, a microarchitecture can simply detect when both source registers are the same for FSGNJ instructions and only read a single copy.

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.W moves the single-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the lower 32 bits of integer register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

FMV.W.X moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register *rs1* to the floating-point register *rd*. The bits are not modified in the

transfer, and in particular, the payloads of non-canonical NaNs are preserved.

The FMV.W.X and FMV.X.W instructions were previously called FMV.S.X and FMV.X.S. The use of W is more consistent with their semantics as an instruction that moves 32 bits without interpreting them. This became clearer after defining NaN-boxing. To avoid disturbing existing code, both the W and S versions will be supported by tools.

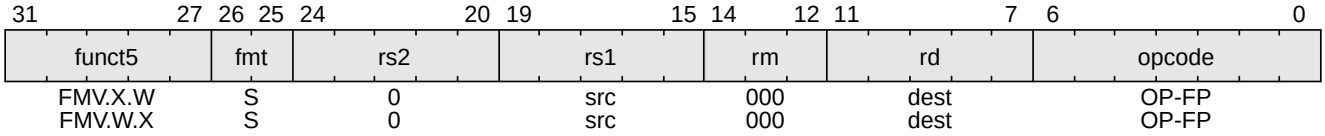


Figure 35. SP floating point move

The base floating-point ISA was defined so as to allow implementations to employ an internal recoding of the floating-point format in registers to simplify handling of subnormal values and possibly to reduce functional unit latency. To this end, the F extension avoids representing integer values in the floating-point registers by defining conversion and comparison operations that read and write the integer register file directly. This also removes many of the common cases where explicit moves between integer and floating-point registers are required, reducing instruction count and critical paths for common mixed-format code sequences.

12.8. Single-Precision Floating-Point Compare Instructions

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

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

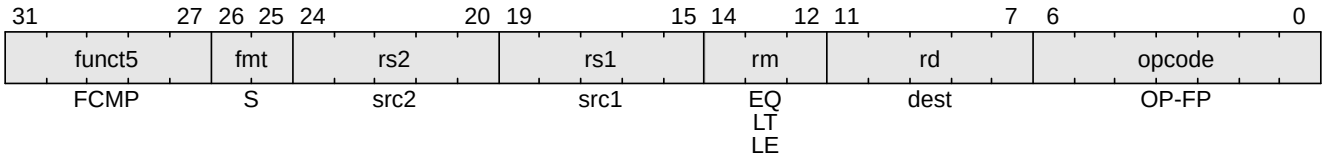


Figure 36. SP floating point compare

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

12.9. 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, “Format of result of FCLASS instruction.”. 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.

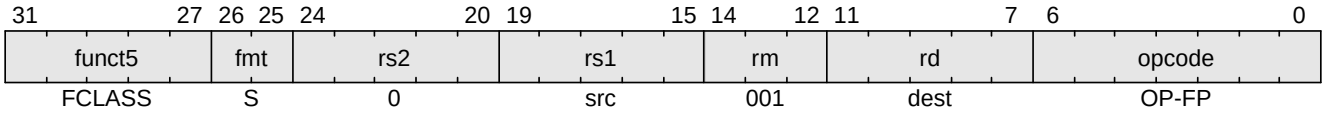


Figure 37. SP floating point classify

Table 11. Format of result of FCLASS instruction.

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

Chapter 13. 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. double-precision, floating point

13.1. D Register State

The D extension widens the 32 floating-point registers, `f0`–` f31`, to 64 bits (FLEN=64 in [fprs-d]. The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section 13.2, "NaN Boxing of Narrower Values".

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.

13.2. 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 (FLn/FSn) and floating-point move instructions (FMV.n.X/FMV.X.n). A narrower n-bit transfer, n<FLEN, into the `f` registers will create a valid NaN-boxed value. A narrower n-bit transfer out of the floating-point registers will transfer the lower n bits of the register ignoring the upper FLEN−n bits.

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

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

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

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

13.3. 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.

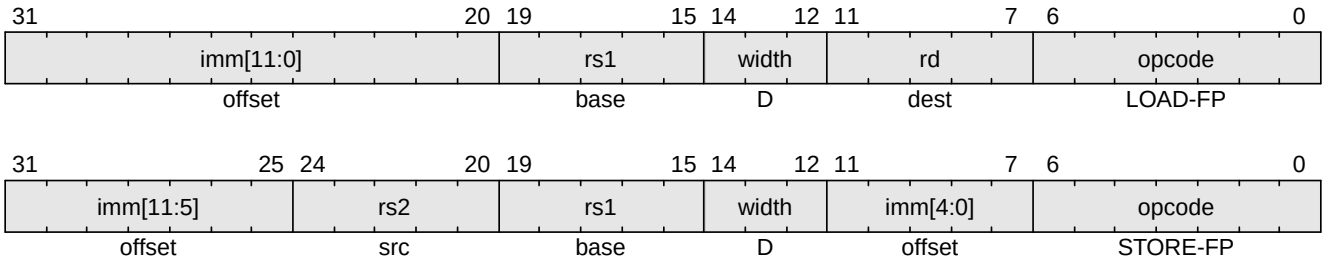


Figure 38. Double-precision load and store

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.

13.4. Double-Precision Floating-Point Computational Instructions

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

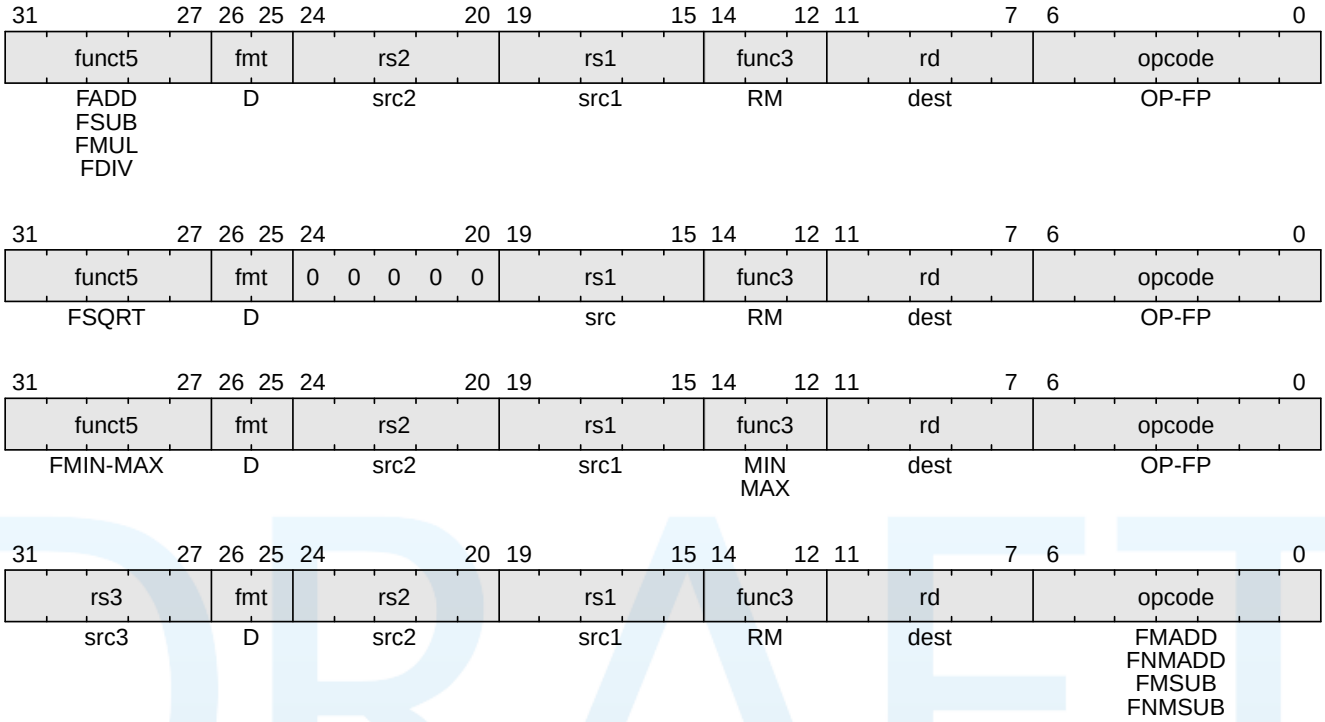


Figure 39. Double-precision float computational

13.5. 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.

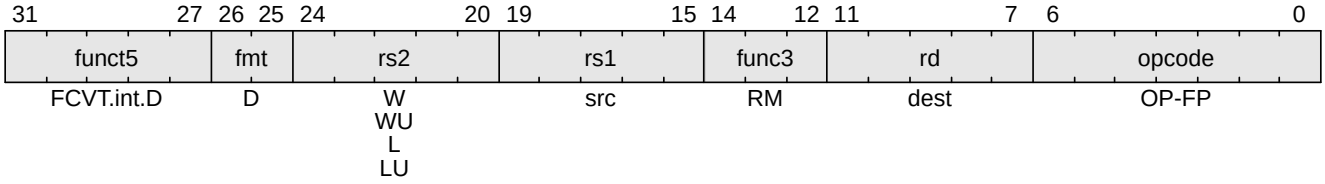


Figure 40. Double-precision float convert and move

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.

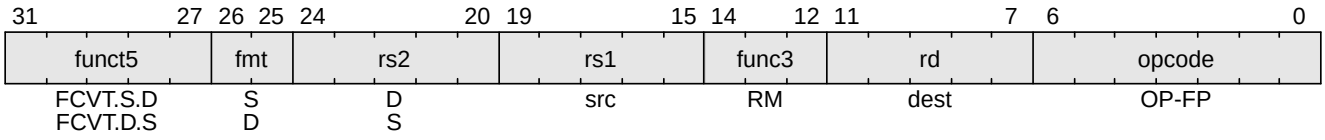


Figure 41. Double-precision FCVT.S.D and FCVT.D.S

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.

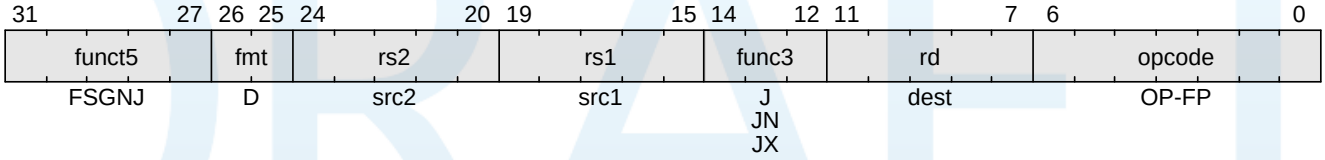


Figure 42. Double-precision sign-injection

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.

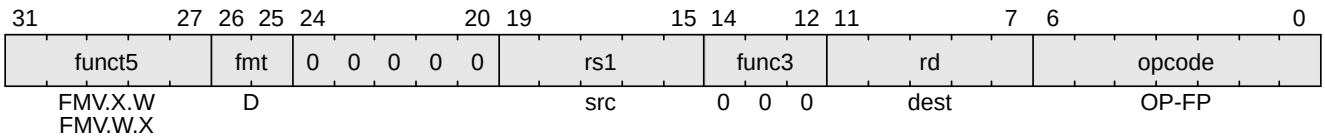


Figure 43. Double-precision float move to rd

Early versions of the RISC-V ISA had additional instructions to allow RV32 systems to transfer between the upper and lower portions of a 64-bit floating-point register and an integer register. However, these would be the only instructions with partial register writes and would add complexity in implementations with recoded floating-point or register renaming, requiring a pipeline read-modify-write sequence. Scaling up to handling quad-precision for RV32 and RV64 would also require additional instructions if they were to follow this pattern. The ISA was defined to reduce the number of explicit int-float register moves, by having conversions and comparisons write results to the appropriate register file, so we expect the benefit of these instructions to be lower than for other ISAs.

We note that for systems that implement a 64-bit floating-point unit including fused multiply-add support and 64-bit floating-point loads and stores, the marginal hardware cost of moving from a 32-bit to a 64-bit integer datapath is low, and a software ABI supporting 32-bit wide address-space and pointers can be used to avoid growth of static data and dynamic memory traffic.

13.6. Double-Precision Floating-Point Compare Instructions

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

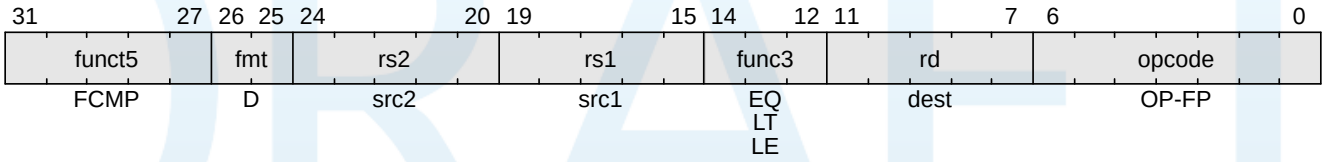


Figure 44. Double-precision float compare

13.7. 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.

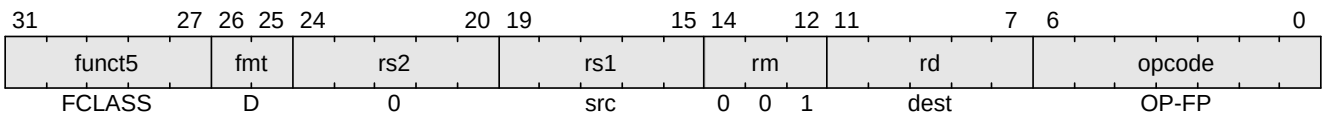


Figure 45. Double-precision float classify

Chapter 14. Q Standard Extension for Quad-Precision Floating-Point, Version 2.2

This chapter describes the Q standard extension for 128-bit quad-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The quad-precision binary floating-point instruction-set extension is named `Q`; it depends on the double-precision floating-point extension D. The floating-point registers are now extended to hold either a single, double, or quad-precision floating-point value (FLEN=128). The NaN-boxing scheme described in Section 13.2, "NaN Boxing of Narrower Values" is now extended recursively to allow a single-precision value to be NaN-boxed inside a double-precision value which is itself NaN-boxed inside a quad-precision value.

14.1. 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.

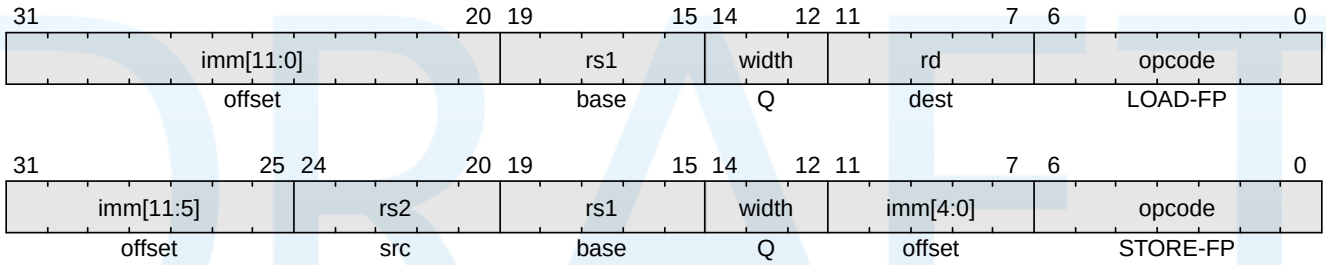


Figure 46. Quad-precision load and store

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.

14.2. Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 12, "Format field encoding."

Table 12. Format field encoding.

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

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

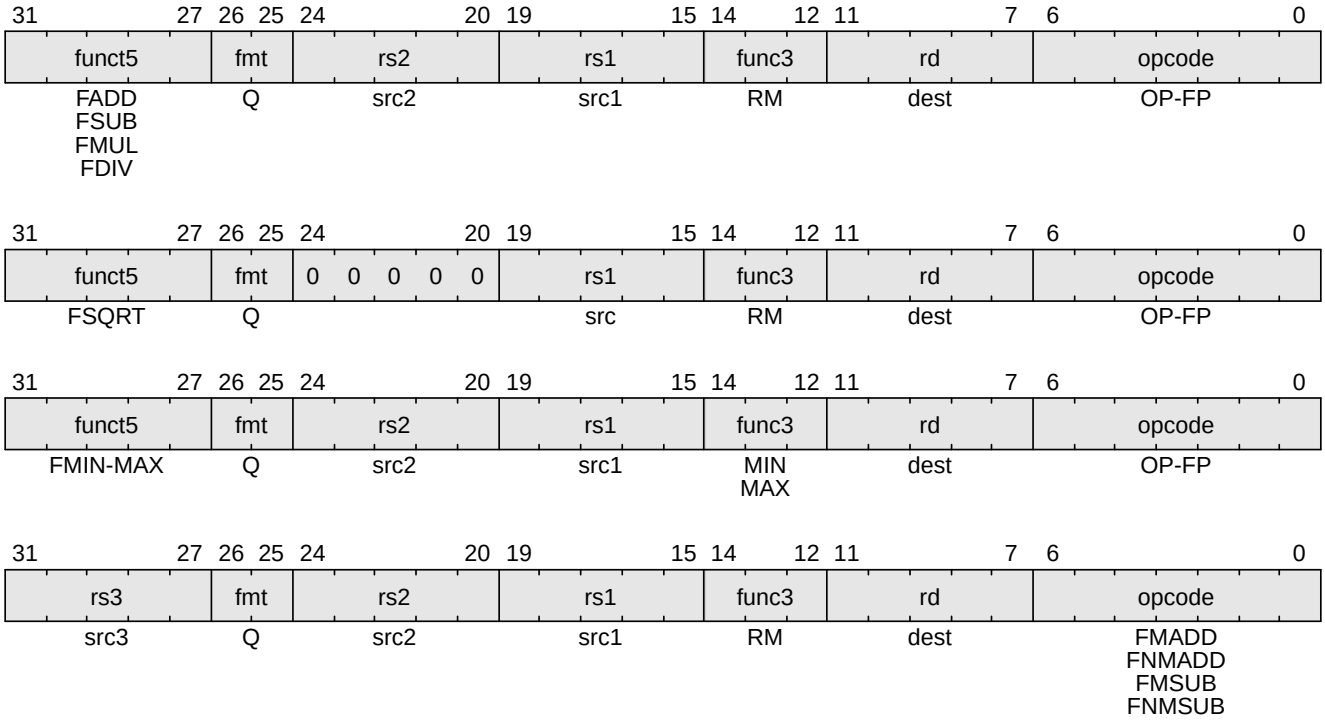


Figure 47. Quad-precision computational

14.3. Quad-Precision Convert and Move Instructions

New floating-point-to-integer and integer-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision-to-integer and integer-to-double-precision conversion instructions. FCVT.W.Q or FCVT.L.Q converts a quad-precision floating-point number to a signed 32-bit or 64-bit integer, respectively. FCVT.Q.W or FCVT.Q.L converts a 32-bit or 64-bit signed integer, respectively, into a quad-precision floating-point number. FCVT.WU.Q, FCVT.LU.Q, FCVT.Q.WU, and FCVT.Q.LU variants convert to or from unsigned integer values. FCVT.L[U].Q and FCVT.Q.L[U] are RV64-only instructions.

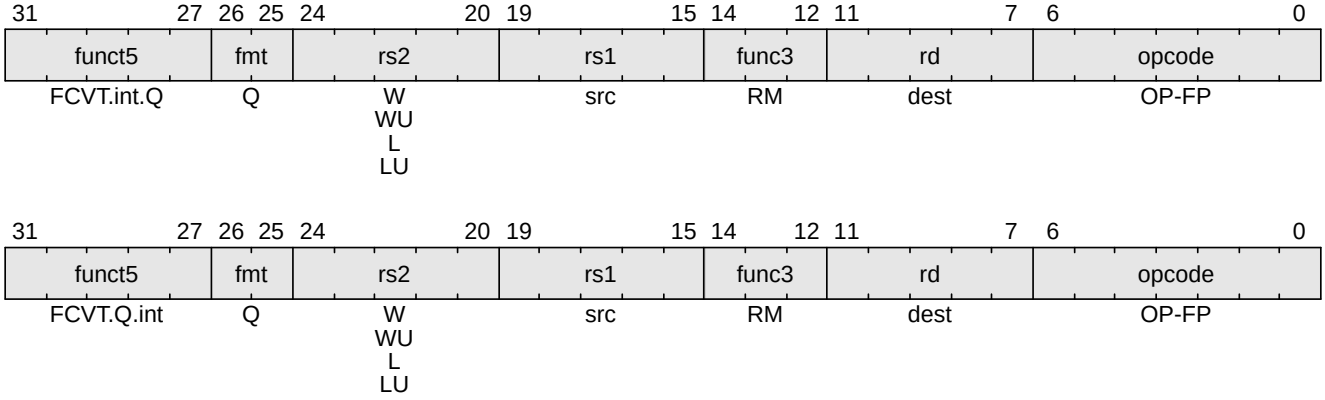


Figure 48. Quad-precision convert and move

New floating-point-to-floating-point conversion instructions are added. These instructions are defined analogously to the double-precision floating-point-to-floating-point conversion instructions. FCVT.S.Q or FCVT.Q.S converts a quad-precision floating-point number to a single-precision floating-point number, or vice-versa, respectively. FCVT.D.Q or FCVT.Q.D converts a quad-precision floating-point number to a double-precision floating-point number, or vice-versa, respectively.

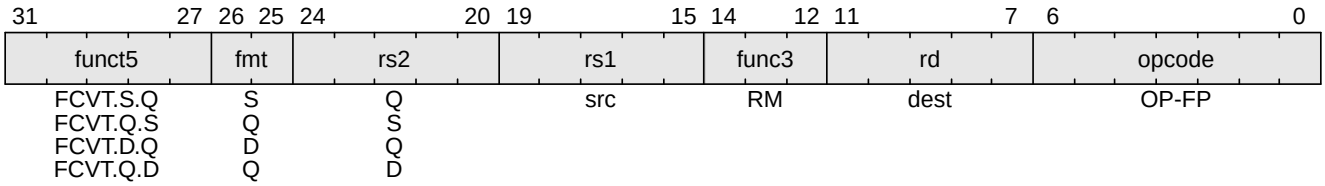


Figure 49. Quad-precision convert and move interchangeably

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.

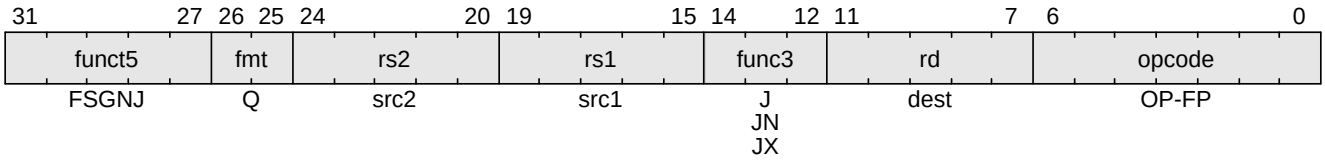


Figure 50. Quad-precision convert and move interchangeably XQ-QX

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.

14.4. Quad-precision floating-Point compare

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

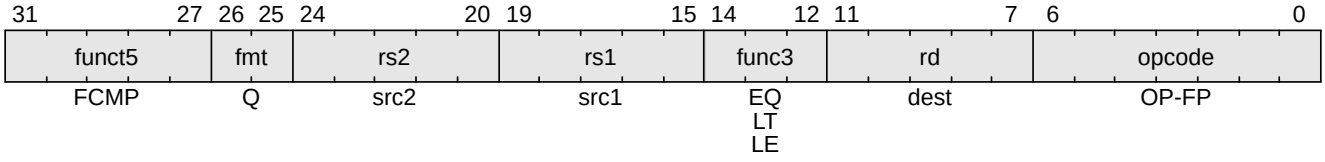


Figure 51. Quad-precision floatinf-point compare

14.5. 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.

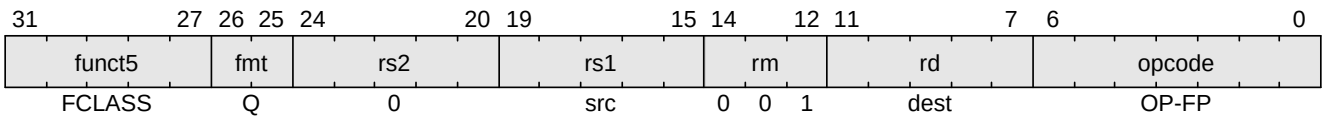


Figure 52. Quad-precision floating point classify

Chapter 15. RVWMO Memory Consistency Model, Version 2.0

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

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

The standard ISA extension for misaligned atomics *Zam* (Chapter 21, *`Zam` Standard Extension for Misaligned Atomics, v0.1*) and the standard ISA extension for total store ordering *Ztso* (Chapter 22, *`Ztso` Standard Extension for Total Store Ordering, v0.1*) 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 and J JIT extensions will need to be incorporated into a future revision as well.

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

15.1. Definition of the RVWMO Memory Model

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

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

15.1.1. Memory Model Primitives

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

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

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 13.3, "Double-Precision Load and Store Instructions" 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 21, *`Zam` Standard Extension for Misaligned Atomics, v0.1*.

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 9.2, "Load-Reserved/Store-Conditional Instructions".

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

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

release-RCsc annotation. An *RCpc annotation* refers to an acquire-RCpc annotation or a release-RCpc annotation. An *RCsc annotation* refers to an acquire-RCsc annotation or a release-RCsc annotation.

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

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

RCpc *annotations are currently only used when implicitly assigned to every memory access per the standard extension* **Ztso** *(Chapter 22, `Ztso` Standard Extension for Total Store Ordering, v0.1). Furthermore, although the ISA does not currently contain native load-acquire or store-release instructions, nor RCpc variants thereof, the RVWMO model itself is designed to be forwards-compatible with the potential addition of any or all of the above into the ISA in a future extension.*

15.1.2. Syntactic Dependencies

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

In the context of defining dependencies, a *register* refers either to an entire general-purpose register, some portion of a CSR, or an entire CSR. The granularity at which dependencies are tracked through CSRs is specific to each CSR and is defined in Section 15.2, "CSR Dependency Tracking Granularity".

Syntactic dependencies are defined in terms of instructions' *source registers*, instructions' *destination registers*, and the way instructions *carry a dependency* from their source registers to their destination registers. This section provides a general definition of all of these terms; however, Section 15.3, "Source and Destination Register Listings" provides a complete listing of the specifics for each instruction.

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

- In the opcode of *i*, *rs1*, *rs2*, or *rs3* is set to *r*
- *i* is a CSR instruction, and in the opcode of *i*, *csr* is set to *r*, unless *i* is CSRRW or CSRRWI and *rd* is set to *x0*
- *r* is a CSR and an implicit source register for *i*, as defined in Section 15.3, "Source and Destination Register Listings"
- *r* is a CSR that aliases with another source register for *i*

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

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

- In the opcode of *i*, *rd* is set to *r*
- *i* is a CSR instruction, and in the opcode of *i*, *csr* is set to *r*, unless *i* is CSRRS or CSRRC and *rs1* is set to *x0* or *i* is CSRRSI or CSRRCI and uimm[4:0] is set to zero.

- *r* is a CSR and an implicit destination register for *i*, as defined in Section 15.3, "Source and Destination Register Listings"
- *r* is a CSR that aliases with another destination register for *i*

Most non-memory instructions *carry a dependency* from each of their source registers to each of their destination registers. However, there are exceptions to this rule; see Section 15.3, "Source and Destination Register Listings".

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

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

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

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

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

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

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

15.1.3. Preserved Program Order

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

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

- Overlapping-Address Orderings:
  1. *b* is a store, and *a* and *b* access overlapping memory addresses

2. a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations
3. a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

- Explicit Synchronization

4. There is a FENCE instruction that orders a before b
5. a has an acquire annotation
6. b has a release annotation
7. a and b both have RCsc annotations
8. a is paired with b

- Syntactic Dependencies

9. b has a syntactic address dependency on a
10. b has a syntactic data dependency on a
11. b is a store, and b has a syntactic control dependency on a

- Pipeline Dependencies

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

15.1.4. Memory Model Axioms

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

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 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.

15.2. CSR Dependency Tracking Granularity

Table 13. Granularities at which syntactic dependencies are tracked through CSRs

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

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

15.3. Source and Destination Register Listings

This section provides a concrete listing of the source and destination registers for each instruction. These listings are used in the definition of syntactic dependencies in Section 15.1.2, "Syntactic Dependencies".

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

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

Key:

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

Table 14. RV32I Base Integer Instruction Set

	Source Registers	Destination Registers	Accumulating CSRs
LUI		*rd*	
AUIPC		*rd*	
JAL		*rd*	

	Source Registers	Destination Registers	Accumulating CSRs
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			

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

Table 15. RV64I Base Integer Instruction Set

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

Table 16. RV32M Standard Extension

	Source Regisers	Destination Registers	Accumulating CSRs
MUL	*rs1*, *rs2*	*rd*	
MULH	*rs1*, *rs2*	*rd*	
MULHSU	*rs1*, *rs2*	*rd*	
MULHU	*rs1*, *rs2*	*rd*	

	Source Regisers	Destination Registers	Accumulating CSRs
DIV	*rs1*, *rs2*	*rd*	
DIVU	*rs1*, *rs2*	*rd*	
REM	*rs1*, *rs2*	*rd*	
REMU	*rs1*, *rs2*	*rd*	

Table 17. RV64M Standard Extension

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

Table 18. RV32A Standard Extension

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

Table 19. RV64A Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs	
LR.D †	*rs1* [A]	*rd*		
SC.D †	*rs1* [A], *rs2* [D]	*rd* [*]		[*]if successful
AMOSWAP.D †	*rs1* [A], *rs2* [D]	*rd*		
AMOADD.D †	*rs1* [A], *rs2* [D]	*rd*		
AMOXOR.D †	*rs1* [A], *rs2* [D]	*rd*		
AMOAND.D †	*rs1* [A], *rs2*[D]	*rd*		

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

Table 20. RV32F Standard Extension

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

Table 21. RV64F Standard Extension

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

Table 22. RV32D Standard Extension

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

Table 23. RV64D Standard Extension

	Source Regsiters	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 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 Table 26, “RVC opcode map instructions.”, a few opcodes are used for different purposes depending on base ISA. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.

Double-precision loads and stores are a significant fraction of static and dynamic instructions, hence the motivation to include them in the RV32C and RV64C encoding.

Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.

Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.

Although reusing opcodes for different purposes for different base ISAs adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISAs. The compressed floating-point load and store variants use the same instruction format with the same register specifiers as the wider integer loads and stores.

RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E, RV64I, or RV128I) or the F and D standard extensions where present. Adopting this constraint has two main benefits:

- Hardware designs can simply expand RVC instructions during decode, simplifying verification and minimizing modifications to existing microarchitectures.
- Compilers can be unaware of the RVC extension and leave code compression to the assembler and linker, although a compression-aware compiler will generally be able to produce better results.

We felt the multiple complexity reductions of a simple one-one mapping between C and base IFD instructions far outweighed the potential gains of a slightly denser encoding that added additional instructions only supported in the C extension, or that allowed encoding of multiple IFD instructions in one C instruction.

It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA.

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch (Buchholz, 1962), developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture (Amdahl et al., 1964) supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 (Thornton, 1965), a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25–30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.

Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to significant complexity in documentation, implementations, and software tools support.

Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size.

16.2. Compressed Instruction Formats

Table Table 26, “RVC opcode map instructions.” 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 [registers] 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 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.

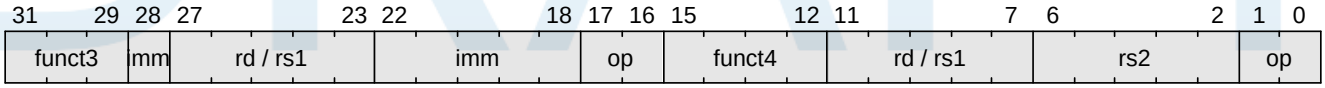


Figure 53. Compressed 16-bit RVC instructions

Table 24. Compressed 16-bit RVC instruction formats.

Format	Meaning	15 14 13 12		11 10 9 8 7	6 5 4 3 2		1 0
CR	Register	funct4		rd/rs1	rs2		op
CI	Immediate	funct3	imm	rd/rs1	imm		op
CSS	Stack-relative Store	funct3	imm		rs2		op
CIW	Wide Immediate	funct3	imm			rd l'	op
CL	Load	funct3	imm	rs1 l'	imm	rd '	op
CS	Store	funct3	imm	rs1 l'	imm	rs2 '	op

Format	Meaning	15 14 13 12		11 10 9 8 7	6 5 4 3 2		1 0
CA	Arithmetic	funct6		rd l'/rs1 l'	funct2	rs2 l'	op
CB	Branch/Arithmetic	funct3	offset	rd l'/rs1 l'	offset		op
CJ	Jump	funct3	jump target				op

Registers specified by the three-bit rs1 l',

rs2 l', and *rd l'* fields of the CIW, CL, CS, CA, and CB formats.

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

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.

16.3.1. Stack-Pointer-Based Loads and Stores

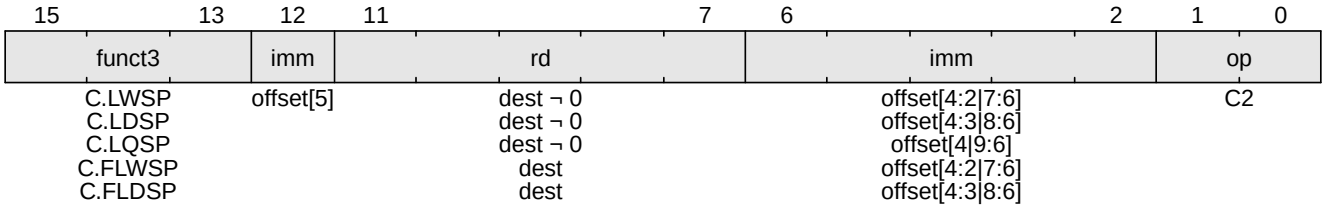


Figure 54. Stack-Pointer-Based Loads and Stores

These instructions use the CI format.

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

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

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

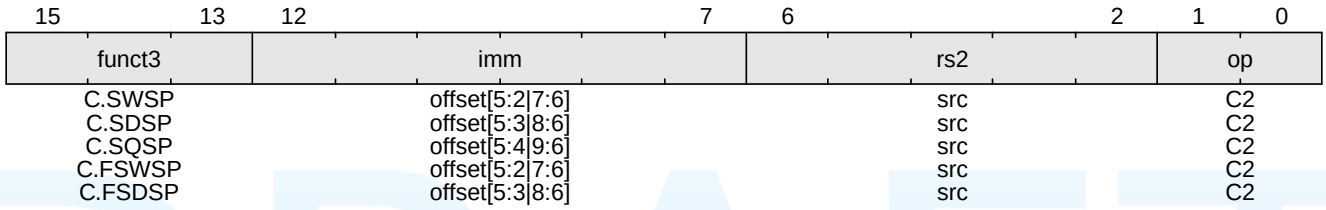


Figure 55. Stack-Pointer-Based Loads and Stores, CSS format

These instructions use the CSS format.

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

C.SDSP is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, *x2*. It expands to *sd rs2, offset(x2)*.

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

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

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

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

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

- *These instructions complicate processor implementations.*
- *For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions.*
- *Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple.*
- *Unlike the rest of the RVC instructions, the compiler would have to be aware of these instructions to both generate the instructions and to allocate registers in an order to maximize the chances of the them being saved and stored, since they would be saved and restored in sequential order.*
- *Simple microarchitectural implementations will constrain how other instructions can be scheduled around the load and store multiple instructions, leading to a potential performance loss.*
- *The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, CA, and CB formats.*

Furthermore, much of the gains can be realized in software by replacing prologue and epilogue code with subroutine calls to common prologue and epilogue code, a technique described in Section 5.6 of .

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.

16.3.2. Register-Based Loads and Stores

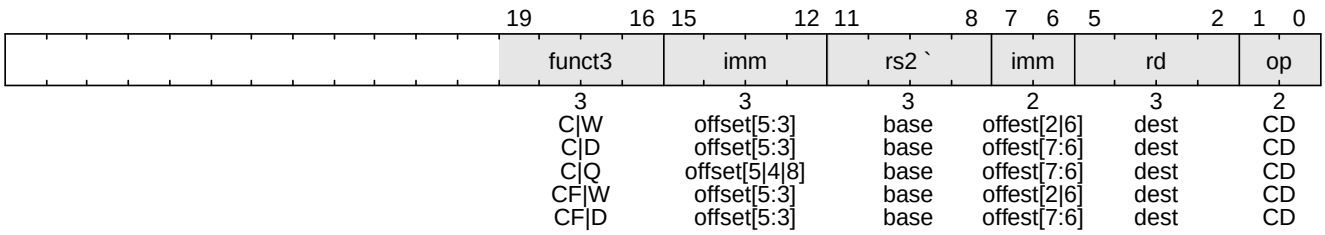


Figure 56. Compressed, register-Based Loads and Stores

These instructions use the CL format.

C.LW loads a 32-bit value from memory into register *rd l'*. It computes an effective address by adding

the *zero*-extended offset, scaled by 4, to the base address in register *rs1 l'*. 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 l'*. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1 l'*. 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 l'*. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1 l'*. 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 l'*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1 l'*. 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 l'*. It computes an effective address by adding the *zero* -extended offset, scaled by 8, to the base address in register *rs1 l'*. It expands to *fld rd, offset(rs1)*.

S@S@S@Y@S@Y
& & & & &
& & & & &
& 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 l'* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1 l'*. It expands to *sw rs2, offset(rs1)*.

C.SD is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2 l'* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1 l'*. It expands to *sd rs2, offset(rs1)*.

C.SQ is an RV128C-only instruction that stores a 128-bit value in register *rs2 l'* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1 l'*. It expands to *sq rs2, offset(rs1)*.

C.FSW is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2 l'* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1 l'*. 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 l'* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1 l'*. 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.

S@L@Y
& &
& &
& 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 ± 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*.

E@T@T@Y
& & &
& & &
& 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 $rs1 \neq x0$; the code point with $rs1 = x0$ is reserved.

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (*pc*+2) to the link register, *x1*. C.JALR expands to *jalr x1, 0(rs1)*. C.JALR is only valid when $rs1 \neq x0$; the code point with $rs1 = x0$ corresponds to the C.EBREAK instruction.

Strictly speaking, C.JALR does not expand exactly to a base RVI instruction as the value added to the PC to form the link address is 2 rather than 4 as in the base ISA, but supporting both offsets of 2 and 4 bytes is only a very minor change to the base microarchitecture.

S@S@S@T@Y
& & & &
& & & &
& 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 ± range. C.BEQZ takes the branch if the value in register *rs1 l'* is zero. It expands to *beq rs1, x0, offset*.

C.BNEZ is defined analogously, but it takes the branch if *rs1 l'* 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.

16.5.1. Integer Constant-Generation Instructions

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

S@W@T@T@Y
& & & &
& & & &
& 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* $rd \neq \{x0, x2\}$, 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.

16.5.2. Integer Register-Immediate Operations

These integer register-immediate operations are encoded in the CI format and perform operations on an integer register and a 6-bit immediate.

S@W@T@T@Y
& & & &
& & & &
& 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.*

@S@K@S@Y
& & &
& & &
& 8 & 3 & 2
C.ADDI4SPN & nzuimm[5:4|9:6|2|3] & dest & C0

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.

S@W@T@T@Y
& & & &
& & & &
& 1 & 5 & 5 & 2
C.SLLI & shamt[5] & dest≠0 & shamt[4:0] & C2

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.

S@W@Y@S@T@Y
& & & & &
& & & & &
& 1 & 2 & 3 & 5 & 2
C.SRLI & shamt[5] & C.SRLI & dest & shamt[4:0] & C1
C.SRAI & shamt[5] & C.SRAI & dest & shamt[4:0] & C1

C.SRLI is a CB-format instruction that performs a logical right shift of the value in register *rd l'* then writes the result to *rd l'*. 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.

S@W@Y@S@T@Y
& & & & &
& & & & &
& 1 & 2 & 3 & 5 & 2
C.ANDI & imm[5] & C.ANDI & dest & imm[4:0] & C1

C.ANDI is a CB-format instruction that computes the bitwise AND of the value in register *rd l'* and the sign-extended 6-bit immediate, then writes the result to *rd l'*. C.ANDI expands to *andi rd, rd, imm*.

16.5.3. Integer Register-Register Operations

E@T@T@Y
& & &
& & &
& 5 & 5 & 2
C.MV & dest≠0 & src≠0 & C2
C.ADD & dest≠0 & src≠0 & C2

These instructions use the CR format.

C.MV copies the value in register *rs2* into register *rd*. C.MV expands into *add rd, x0, rs2*. C.MV is only valid when $rs2 \neq x0$; the code points with $rs2 = x0$ correspond to the C.JR instruction. The code points with $rs2 \neq x0$ and $rd = x0$ are HINTs.

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

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

M@S@Y@S@Y
& & & &
& & & &
& 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 l'* and *rs2 l'*, then writes the result to register *rd l'*. C.AND expands into *and rd, rd, rs2*.

C.OR computes the bitwise OR of the values in registers *rd l'* and *rs2 l'*, then writes the result to register *rd l'*. C.OR expands into *or rd', rd', rs2'*.

C.XOR computes the bitwise XOR of the values in registers *rd l'* and *rs2 l'*, then writes the result to register *rd l'*. C.XOR expands into *xor rd', rd', rs2'*.

C.SUB subtracts the value in register *rs2 l'* from the value in register *rd l'*, then writes the result to register *rd l'*. C.SUB expands into *sub rd', rd', rs2'*.

C.ADDW is an RV64C/RV128C-only instruction that adds the values in registers *rd l'* and *rs2 l'*, then sign-extends the lower 32 bits of the sum before writing the result to register *rd l'*. C.ADDW expands into *addw rd', rd', rs2'*.

C.SUBW is an RV64C/RV128C-only instruction that subtracts the value in register *rs2 l'* from the value in register *rd l'*, then sign-extends the lower 32 bits of the difference before writing the result to register *rd l'*. 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.

16.5.4. Defined Illegal Instruction

SW@T@T@Y
& & & &
& & & &
& 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.

16.5.5. NOP Instruction

SW@T@T@Y
& & & &

& & & &
& 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.

16.5.6. Breakpoint Instruction

E@U@Y
& &
& &
& 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 9.3, "Eventual Success of Store-Conditional Instructions", 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 [rv32i-hints], 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 25, "RVC HINT instructions." 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.

Table 25. RVC HINT instructions.

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	31	
C.SRLI64	RV32 and RV64 only	8	
C.SRAI64	RV32 and RV64 only	8	

16.8. RVC Instruction Set Listings

Table 26, "RVC opcode map instructions." shows a map of the major opcodes for RVC. Each row of the table corresponds to one quadrant of the encoding space. The last quadrant, which has the two least-significant bits set, corresponds to instructions wider than 16 bits, including those in the base ISAs. Several instructions are only valid for certain operands; when invalid, they are marked either *RES* to indicate that the opcode is reserved for future standard extensions; *Custom* to indicate that the opcode is designated for custom extensions; or *HINT* to indicate that the opcode is reserved for microarchitectural hints Section 16.7, "HINT Instructions".

Table 26. RVC opcode map instructions.

inst[15:13]										inst[1:0]
00	000	001	010	011	100	101	110	111		ADDI4SPN
FLD FLD LQ	LW	FLW LD LD	*Reserved*	FSD FSD SQSW	SW	FSW SD SD	RV32 RV64 RV128	01	ADDI	JAL ADDIW ADDIW

LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128	I0	SLLI	FLDSP FLDSP LDSP	LWSP

Table 27, “Instruction listing for RVC, Quadrant 0”, Table 28, “Instruction listing for RVC, Quadrant 1”, and Table 29, “Instruction listing for RVC, Quadrant 2” list the RVC instructions.

Table 27. Instruction listing for RVC, Quadrant 0

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 l'	uimm[7:6]	rd'	00	C.FLD *(RV32/64)*
001	uimm[5:4|8]	rs1 l'	uimm[7:6]	rd'	00	C.LQ *(RV128)*
010	uimm[5:3]	rs1 l'	uimm[2|6]	rd'	00	C.LW
011	uimm[5:3]	rs1 l'	uimm|6]	rd'	00	C.FLW *(RV32)*
011	uimm[5:3]	rs1 l'	uimm[7:6]	rd'	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1 l'	uimm[7:6]	rs2'	00	C.FSD *(RV32/64)*
101	uimm[5:4|8]	rs1 l'	uimm[7:6]	rs2'	00	C.SQ *(RV128)*
110	uimm[5:3]	rs1 l'	uimm[2|6]	rs2'	00	C.SW
111	uimm[5:3]	rs1 l'	uimm[2|6]	rs2'	00	C.FSW *(RV32)*
111	uimm[5:3]	rs1 l'	uimm[7:6]	rs2'	00	C.SD *(RV64/128)*

Table 28. Instruction listing for RVC, Quadrant 1

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]	2		nzimm[4|6|8:7|5]	01	C.ADDI16SP *(RES, nzimm=0)*
011	nzimm[17]	rd ≠{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 Custom, nzuimm[5]=1)*
100	0	00	rs1'/rd l'	0	01	C.SRLI64 *(RV128; RV32/64 HINT)*

15 14 13	12	11 10 9 8 7		6 5 4 3 2		1 0	
100	nzuimm[5]	01	rs1'/rd'	nzuimm[4:0]		01	C.SRAI *(RV32 Custom, nzuimm[5]=1)*
100	0	01	rs1'/rd'	0		01	C.SRAI64 *(RV128; RV32/64 HINT)*
100	imm[5]	10	rs1 l'/rd'	imm[4:0]		01	*C.ANDI*
100	0	11	rs1'/rd'	00	rs2'	01	*C.SUB*
100	0	11	rs1'/rd'	01	rs2 l'	01	*C.XOR*
100	0	11	rs1'/rd'	10	rs2 l'	01	*C.OR*
100	0	11	rs1'/rd'	11	rs2 l'	01	*C.AND*
100	1	11	rs1'/rd'	00	rs2 l'	01	*C.SUBW (RV64/128; RV32 RES)*
100	1	11	rs1'/rd'	01	rs2 l'	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 29. Instruction listing for RVC, Quadrant 2

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 Custom, 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)*

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
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)*

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 ISAs as they are not useful in all domains and can add additional complexity or instruction formats to supply all needed operands.

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

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. 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.

Chapter 20. V Standard Extension for Vector Operations, Version 0.7

The current working group draft is hosted at `github.com/riscv/riscv-v-spec`.

The base vector extension is intended to provide general support for data-parallel execution within the 32-bit instruction encoding space, with later vector extensions supporting richer functionality for certain domains.

Chapter 21. Zam Standard Extension for Misaligned Atomics, v0.1

This chapter defines the Zam extension, which extends the A extension by standardizing support for misaligned atomic memory operations (AMOs). On platforms implementing Zam, misaligned AMOs need only execute atomically with respect to other accesses (including non-atomic loads and stores) to the same address and of the same size. More precisely, execution environments implementing Zam are subject to the following axiom:

21.1. Atomicity Axiom for misaligned atomics

If r and w are paired misaligned load and store instructions from a hart h with the same address and of the same size, then there can be no store instruction s from a hart other than h with the same address and of the same size as r and w such that a store operation generated by s lies in between memory operations generated by r and w in the global memory order. Furthermore, there can be no load instruction l from a hart other than h with the same address and of the same size as r and w such that a load operation generated by l lies between two memory operations generated by r or by w in the global memory order.

This restricted form of atomicity is intended to balance the needs of applications which require support for misaligned atomics and the ability of the implementation to actually provide the necessary degree of atomicity.

Aligned instructions under Zam continue to behave as they normally do under RVWMO.

The intention of Zam is that it can be implemented in one of two ways:

1. On hardware that natively supports atomic misaligned accesses to the address and size in question (e.g., for misaligned accesses within a single cache line): by simply following the same rules that would be applied for aligned AMOs.
2. On hardware that does not natively support misaligned accesses to the address and size in question: by trapping on all instructions (including loads) with that address and size and executing them (via any number of memory operations) inside a mutex that is a function of the given memory address and access size. AMOs may be emulated by splitting them into separate load and store operations, but all preserved program order rules (e.g., incoming and outgoing syntactic dependencies) must behave as if the AMO is still a single memory operation.

Chapter 22. Ztso Standard Extension for Total Store Ordering, v0.1

This chapter defines the Ztso extension for the RISC-V Total Store Ordering (RVTSO) memory consistency model. RVTSO is defined as a delta from RVWMO, which is defined in Section 15.1, "Definition of the RVWMO Memory Model".

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

RVTSO makes the following adjustments to RVWMO:

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

These rules render all PPO rules except Section 2.7, "Memory Ordering Instructions"–[rcsc] redundant. They also make redundant any non-I/O fences that do not have both PW and SR set. Finally, they also imply that no memory operation will be reordered past an AMO in either direction.

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

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

Chapter 23. RV32/64G Instruction Set Listings

One goal of the RISC-V project is that it be used as a stable software development target. For this purpose, we define a combination of a base ISA (RV32I or RV64I) plus selected standard extensions (IMAFD, Zicsr, Zifencei) as a `general-purpose` ISA, and we use the abbreviation G for the IMAFDZicsr_Zifencei combination of instruction-set extensions. This chapter presents opcode maps and instruction-set listings for RV32G and RV64G.

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

[opcodemap] shows a map of the major opcodes for RVG. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. Opcodes marked as *reserved* should be avoided for custom instruction-set extensions as they might be used by future standard extensions. Major opcodes marked as *custom-0* and *custom-1* will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked *custom-2/rv128* and *custom-3/rv128* are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions in RV32 and RV64.

We believe RV32G and RV64G provide simple but complete instruction sets for a broad range of general-purpose computing. The optional compressed instruction set described in Chapter 16, *C Standard Extension for Compressed Instructions, Version 2.0* can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

As we move beyond IMAFDC into further instruction-set extensions, the added instructions tend to be more domain-specific and only provide benefits to a restricted class of applications, e.g., for multimedia or security. Unlike most commercial ISAs, the RISC-V ISA design clearly separates the base ISA and broadly applicable standard extensions from these more specialized additions. [extensions] has a more extensive discussion of ways to add extensions to the RISC-V ISA.

31 27	26 25	24 21	20 19	15 14	12 11	7 6	0


RISC-V