The RISC-V Instruction Set Manual

Volume I: User-Level ISA

Document Version 2.3-draft

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
andrew@sifive.com, krste@berkeley.edu

December 16, 2017

Contributors to all versions of the spec in alphabetical order (please contact editors to suggest corrections): Krste Asanović, Rimas Avižienis, Jacob Bachmeyer, Christopher F. Batten, Allen J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, Chuanhua Chang, David Chisnall, Paul Clayton, Palmer Dabbelt, Roger Espasa, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David Horner, Olof Johansson, Ben Keller, Yunsup Lee, Joseph Myers, Rishiyur Nikhil, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Colin Schmidt, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Ray VanDeWalker, Megan Wachs, Andrew Waterman, Robert Watson, and Reinoud Zandijk.

Please cite as: "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

Preface

This is **a draft of** version 2.3 of the document describing the RISC-V user-level architecture. The document contains the following versions of the RISC-V ISA modules:

Base	Version	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	0.2	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:

- Improvements to the description and commentary.
- 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.
- LR, SC, and AMO instructions are now permitted, but not required, to support misaligned addresses, in which case regular loads and stores to misaligned addresses are also atomic.

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	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	0.2	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 [1].
- 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 [36], 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 [35] of this ISA specification.

- The ISA has been divided into an integer base with several standard extensions.
- The instruction formats have been rearranged to make immediate encoding more efficient.
- The base ISA has been defined to have a little-endian memory system, with big-endian or bi-endian as non-standard variants.
- Load-Reserved/Store-Conditional (LR/SC) instructions have been added in the atomic instruction extension.
- AMOs and LR/SC can support the release consistency model.
- The FENCE instruction provides finer-grain memory and I/O orderings.
- An AMO for fetch-and-XOR (AMOXOR) has been added, and the encoding for AMOSWAP has been changed to make room.
- The AUIPC instruction, which adds a 20-bit upper immediate to the PC, replaces the RDNPC instruction, which only read the current PC value. This results in significant savings for position-independent code.
- The JAL instruction has now moved to the U-Type format with an explicit destination register, and the J instruction has been dropped being replaced by JAL with *rd*=`x0`. This removes the only instruction with an implicit destination register and removes the J-Type instruction format from the base ISA. There is an accompanying reduction in JAL reach, but a significant reduction in base ISA complexity.
- The static hints on the JALR instruction have been dropped. The hints are redundant with the *rd* and *rs1* register specifiers for code compliant with the standard calling convention.
- The JALR instruction now clears the lowest bit of the calculated target address, to simplify hardware and to allow auxiliary information to be stored in function pointers.
- The MFTX.S and MFTX.D instructions have been renamed to FMV.X.S and FMV.X.D, respectively. Similarly, MXTF.S and MXTF.D instructions have been renamed to FMV.S.X and FMV.D.X, respectively.
- The MFFSR and MTFSR instructions have been renamed to FRCSR and FSCSR, respectively. FRRM, FSRM, FRFLAGS, and FSFLAGS instructions have been added to individually access the rounding mode and exception flags subfields of the `fcsr`.
- The FMV.X.S and FMV.X.D instructions now source their operands from *rs1*, instead of *rs2*. This change simplifies datapath design.
- FCLASS.S and FCLASS.D floating-point classify instructions have been added.
- A simpler NaN generation and propagation scheme has been adopted.
- For RV32I, the system performance counters have been extended to 64-bits wide, with separate read access to the upper and lower 32 bits.
- Canonical NOP and MV encodings have been defined.
- Standard instruction-length encodings have been defined for 48-bit, 64-bit, and >64-bit instructions.
- Description of a 128-bit address space variant, RV128, has been added.
- Major opcodes in the 32-bit base instruction format have been allocated for user-defined custom extensions.

- A typographical error that suggested that stores source their data from *rd* has been corrected to refer to *rs2*.

Contents

Chapter 1

Introduction

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

- A completely *open* ISA that is freely available to academia and industry.
- A *real* ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- An ISA separated into a *small* base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development.
- Support for the revised 2008 IEEE-754 floating-point standard [14].
- An ISA supporting extensive user-level 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 supervisor-level and hypervisor-level ISA designs.

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

The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I [23], RISC-II [15], SOAR [32], and SPUR [18] 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.

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

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

- **Commercial ISAs are proprietary.** *Except for SPARC V8, which is an open IEEE standard [2], most owners of commercial ISAs carefully guard their intellectual property and do not welcome freely available competitive implementations. This is much less of an issue for academic research and teaching using only software simulators, but has been a major concern for groups wishing to share actual RTL implementations. It is also a major concern for entities who do not want to trust the few sources of commercial ISA implementations, but who are prohibited from creating their own clean room implementations. We cannot guarantee that all RISC-V implementations will be free of third-party patent infringements, but we can guarantee we will not attempt to sue a RISC-V implementor.*
- **Commercial ISAs are only popular in certain market domains.** *The most obvious examples at time of writing are that the ARM architecture is not well supported in the server space, and the Intel x86 architecture (or for that matter, almost every other architecture) is not well supported in the mobile space, though both Intel and ARM are attempting to enter each other's market segments. Another example is ARC and Tensilica, which provide extensible cores but are focused on the embedded space. This market segmentation dilutes the benefit of supporting a particular commercial ISA as in practice the software ecosystem only exists for certain domains, and has to be built for others.*
- **Commercial ISAs come and go.** *Previous research infrastructures have been built around commercial ISAs that are no longer popular (SPARC, MIPS) or even no longer in production (Alpha). These lose the benefit of an active software ecosystem, and the lingering intellectual property issues around the ISA and supporting tools interfere with the ability of interested third parties to continue supporting the ISA. An open ISA might also lose popularity, but any interested party can continue using and developing the ecosystem.*
- **Popular commercial ISAs are complex.** *The dominant commercial ISAs (x86 and ARM) are both very complex to implement in hardware to the level of supporting common software stacks and operating systems. Worse, nearly all the complexity is due to bad, or at least outdated, ISA design decisions rather than features that truly improve efficiency.*
- **Commercial ISAs alone are not enough to bring up applications.** *Even if we expend the effort to implement a commercial ISA, this is not enough to run existing applications for that ISA. Most applications need a complete ABI (application binary interface) to run, not just the user-level ISA. Most ABIs rely on libraries, which in turn rely on operating system support. To run an existing operating system requires implementing the supervisor-level ISA and device interfaces expected by the OS. These are usually much less well-specified and considerably more complex to implement than the user-level ISA.*

- **Popular commercial ISAs were not designed for extensibility.** *The dominant commercial ISAs were not particularly designed for extensibility, and as a consequence have added considerable instruction encoding complexity as their instruction sets have grown. Companies such as Tensilica (acquired by Cadence) and ARC (acquired by Synopsys) have built ISAs and toolchains around extensibility, but have focused on embedded applications rather than general-purpose computing systems.*
- **A modified commercial ISA is a new ISA.** *One of our main goals is to support architecture research, including major ISA extensions. Even small extensions diminish the benefit of using a standard ISA, as compilers have to be modified and applications rebuilt from source code to use the extension. Larger extensions that introduce new architectural state also require modifications to the operating system. Ultimately, the modified commercial ISA becomes a new ISA, but carries along all the legacy baggage of the base ISA.*

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

We are far from the first to contemplate an open ISA design suitable for hardware implementation. We also considered other existing open ISA designs, of which the closest to our goals was the OpenRISC architecture [22]. We decided against adopting the OpenRISC ISA for several technical reasons:

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

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

The RISC-V manual is structured in two volumes. This volume covers the user-level ISA design, including optional ISA extensions. The second volume provides the privileged architecture.

In this user-level manual, we aim to remove any dependence on particular microarchitectural features or on privileged architecture details. This is both for clarity and to allow maximum flexibility for alternative implementations.

1.1 RISC-V ISA Overview

The 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 ISA is very similar to that of the early

RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. The 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 supervisor-level operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are two primary base integer variants, RV32I and RV64I, described in Chapters 2 and 4, which provide 32-bit or 64-bit user-level address spaces respectively. Hardware implementations and operating systems might provide only one or both of RV32I and RV64I for user programs. Chapter 3 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers. Chapter 5 describes a future RV128I variant of the base integer instruction set supporting a flat 128-bit user address space. 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 base integer ISA may be subset by a hardware implementation, but opcode traps and software emulation by a more privileged layer must then be used to implement functionality not provided by hardware.

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.

RISC-V has been designed to support extensive customization and specialization. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined. We divide RISC-V instruction-set extensions into *standard* and *non-standard* extensions. Standard extensions should be generally useful and should not conflict with other standard extensions. Non-standard extensions may be highly specialized, or may conflict with other standard or non-standard extensions. Instruction-set extensions may provide slightly different functionality depending on the width of the base integer instruction set. Chapter 21 describes various ways of extending the RISC-V ISA. We have also developed a naming convention for RISC-V base instructions and instruction-set extensions, described in detail in Chapter 22.

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, and is mandatory for all RISC-V implementations. 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. An integer base plus these four standard extensions ("IMAFD") is given the abbreviation "G" and provides a general-purpose scalar instruction set. RV32G and RV64G are currently the default target of our compiler toolchains. Later chapters describe these and other planned standard RISC-V extensions.

Beyond the base integer ISA and the standard extensions, 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.

With the 2.0 release of the user ISA specification, we intend the "RV32IMAFD" and "RV64IMAFD"base and standard extensions (aka. "RV32G" and "RV64G") to remain constant for future development.

1.2 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 12 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 ILEN to refer to the maximum instruction length supported by an implementation, which is always a multiple of 16 bits. For implementations supporting only the base instruction set, ILEN is 32 bits. Implementations supporting longer instructions have larger values of ILEN. ILEN is implied from the set of extensions implemented, or can be explicitly defined in the platform configuration if an implementation is designed to support an extension that uses longer instructions via software emulation but does not actually decode longer instructions in hardware.

Figure 1.1 illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to `11`. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to `00`, `01`, or `10`. Standard instruction-set extensions encoded with more than 32 bits have additional low-order bits set to `1`, with the conventions for 48-bit and 64-bit lengths shown in Figure 1.1. Instruction lengths between 80 bits and 176 bits are encoded using a 3-bit field in bits [14:12] giving the number of 16-bit words in addition to the first 5×16-bit words. The encoding with bits [14:12] set to `111` is reserved for future longer instruction encodings. The encodings with bits [15:0] all zeros or all ones are illegal.

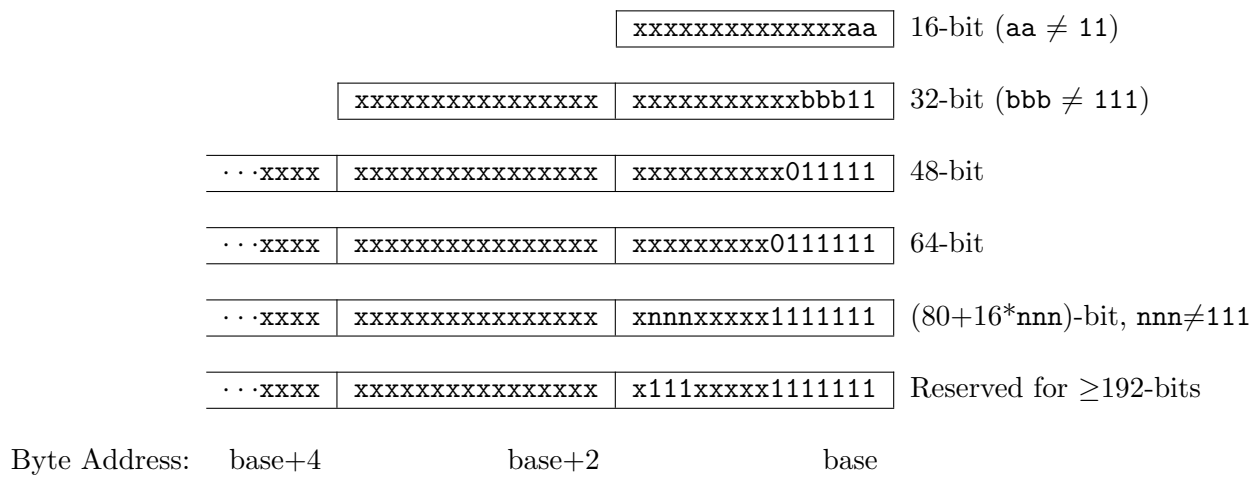


Figure 1.1: RISC-V instruction length encoding.

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 G 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 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 21, an implementation that does not require support for the standard compressed instruction extension can map 3 additional 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.

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.

The base RISC-V ISA has a little-endian memory system, but non-standard variants can provide a big-endian or bi-endian memory system. Instructions are stored in memory with each 16-bit parcel stored in a memory halfword according to the implementation's natural 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, i.e., instructions are always stored in a little-endian sequence of parcels regardless of the memory system endianness. The code sequence in Figure 1.2 will store a 32-bit instruction to memory correctly regardless of memory system endianness.

```
// Store 32-bit instruction in x2 register to location pointed to by x3.
sh   x2, 0(x3)    // Store low bits of instruction in first parcel.
srli x2, x2, 16   // Move high bits down to low bits, overwriting x2.
sh   x2, 2(x3)    // Store high bits in second parcel.
```

Figure 1.2: Recommended code sequence to store 32-bit instruction from register to memory. Operates correctly on both big- and little-endian memory systems and avoids misaligned accesses when used with variable-length instruction-set extensions.

We 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 so we leave open the possibility of non-standard big-endian or bi-endian systems.

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

1.3 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 thread. We use the term *trap* to refer to the synchronous transfer of control to a trap handler caused by an exceptional condition occurring within a RISC-V thread. Trap handlers usually execute in a more privileged environment.

We use the term *interrupt* to refer to an external event that occurs asynchronously to the current RISC-V thread. When an interrupt that must be serviced occurs, some instruction is selected to receive an interrupt exception and subsequently experiences a trap.

The instruction descriptions in following chapters describe conditions that raise an exception during execution. Whether and how these are converted into traps is dependent on the execution environment, though the expectation is that most environments will take a *precise* trap when an exception is signaled (except for floating-point exceptions, which, in the standard floating-point extensions, do not cause traps).

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

Chapter 2

RV32I Base Integer Instruction Set, Version 2.0

This chapter describes version 2.0 of the RV32I base integer instruction set. Much of the commentary also applies to the RV64I variant.

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 47 unique instructions, though a simple implementation might cover the eight ECALL/EBREAK/CSRR instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE and FENCE.I instructions as NOPs, reducing hardware instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).*

2.1 Programmers' Model for Base Integer Subset

Figure 2.1 shows the user-visible state for the base integer subset. There are 31 general-purpose registers `x1`–`x31`, which hold integer values. Register `x0` is hardwired to the constant 0. There is no hardwired subroutine return address link register, but the standard software calling convention uses register `x1` to hold the return address on a call. For RV32, the `x` registers are 32 bits wide, and for RV64, they are 64 bits wide. This document uses the term XLEN to refer to the current width of an `x` register in bits (either 32 or 64).

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

The number of available architectural registers can have large impacts on code size, performance, and energy consumption. Although 16 registers would arguably be sufficient for an integer ISA running compiled code, it is impossible to encode a complete ISA with 16 registers in 16-bit instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. We wanted to avoid intermediate instruction sizes (such as Xtensa's 24-bit instructions) to simplify base hardware implementations, and once

a 32-bit instruction size was adopted, it was straightforward to support 32 integer registers. A larger number of integer registers also helps performance on high-performance code, where there can be extensive use of loop unrolling, software pipelining, and cache tiling.

For these reasons, we chose a conventional size of 32 integer registers for the base ISA. Dynamic register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers [31]. The optional compressed 16-bit instruction format mostly only accesses 8 registers and hence can provide a dense instruction encoding, while additional instruction-set extensions could support a much larger register space (either flat or hierarchical) if desired.

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

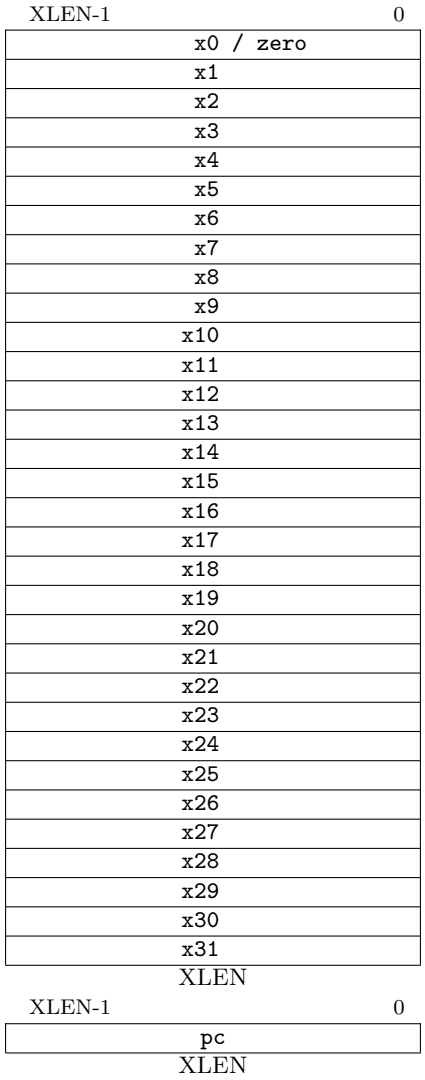


Figure 2.1: RISC-V user-level base integer register state.

2.2 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch 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.

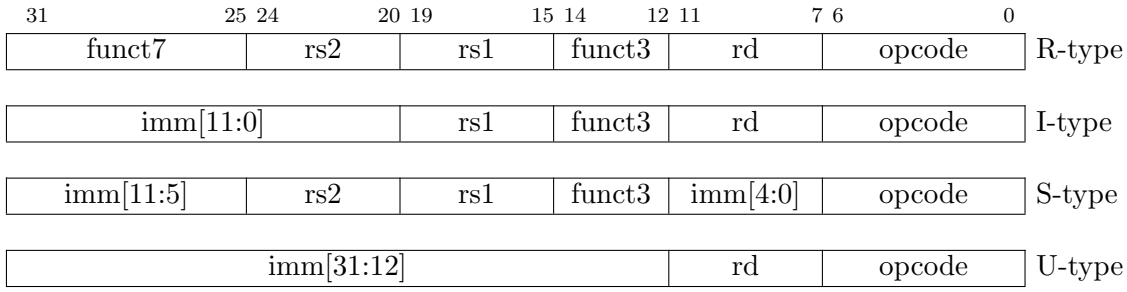


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

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 (Section 2.7), 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.

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 [18]).

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

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

2.3 Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 2.3.

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

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

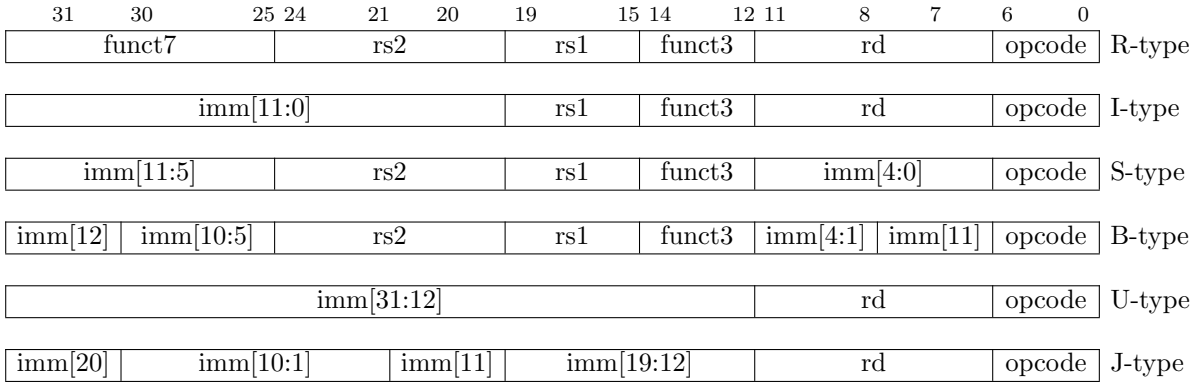


Figure 2.3: RISC-V base instruction formats showing immediate variants.

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

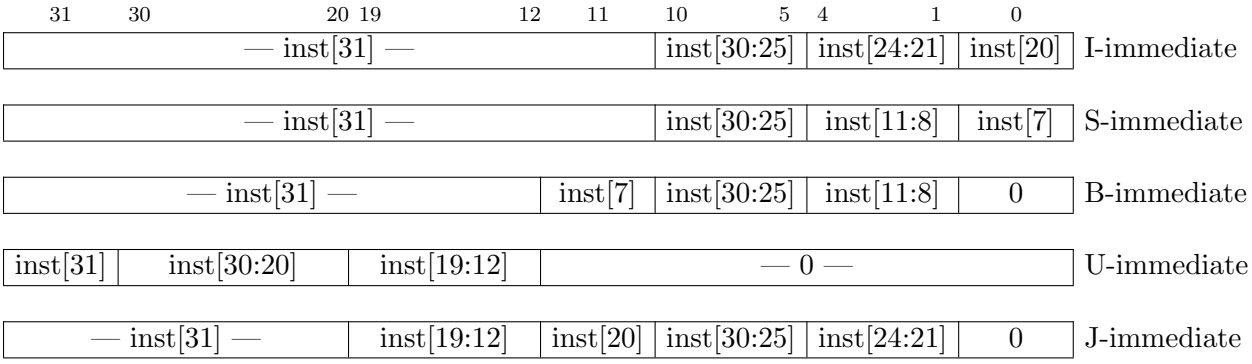


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Sign-extension is one of the most critical operations on immediates (particularly in RV64I), 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 RV64, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

Integer Register-Immediate Instructions

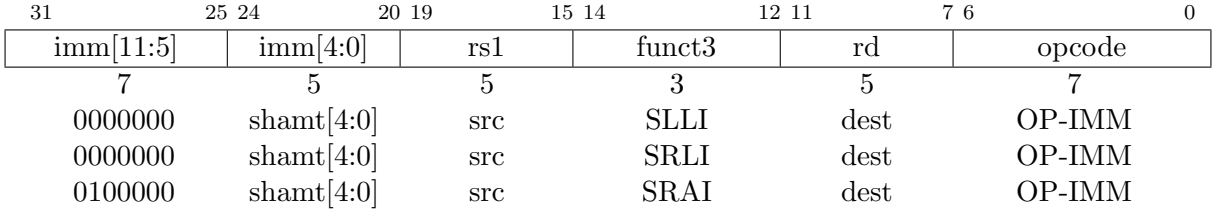
31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

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

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

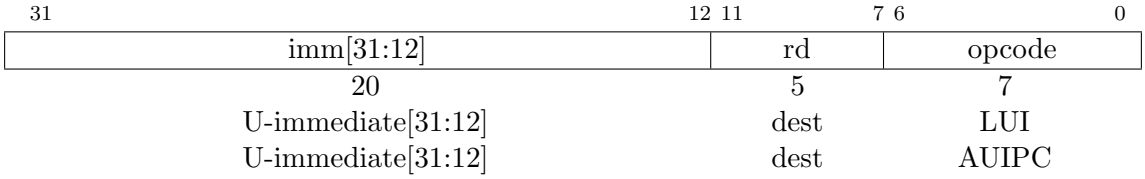
similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number). Note, SLTIU *rd*, *rs1*, 1 sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudo-op SEQZ *rd, rs*).

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 pseudo-instruction NOT *rd, rs*).



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

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



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

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of 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 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the `pc`, then places the result in register *rd*.

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 PC, it might cause pipeline breaks in simpler microarchitectures or pollute the BTB structures in more complex microarchitectures.

Integer Register-Register Operations

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

type of operation.

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

ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination. 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 pseudo-op SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

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

NOP Instruction

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

The NOP instruction does not change any user-visible state, except for advancing the `pc`. 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.

2.5 Control Transfer Instructions

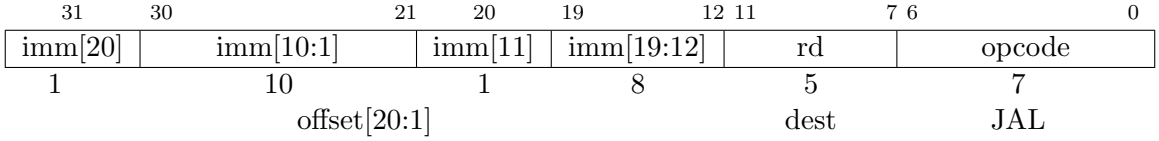
RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches. Control transfer instructions in RV32I do *not* have architecturally visible delay slots.

Unconditional Jumps

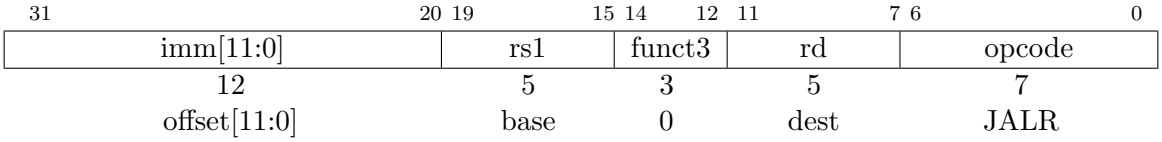
The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the `pc` to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following 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 pseudo-op J) are encoded as a JAL with *rd*=`x0`.



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



The unconditional jump instructions all use PC-relative addressing to help support 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 2 KiB or highest 2 KiB address region from anywhere in the address space, which could be used to implement fast calls to a small runtime library.*

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

Instruction fetch 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*=`x1`/`x5`. JALR instructions should push/pop a RAS as shown in the Table 2.1.

rd	*rs1*	*rs1=rd*	RAS action
!*link*	!*link*	-	none
!*link*	*link*	-	pop
link	!*link*	-	push
link	*link*	0	pop, then push
link	*link*	1	push

Table 2.1: Return-address stack prediction hints encoded in register specifiers used in the instruction. In the above, *link* is true when the register is either `x1` or `x5`.

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, ra, imm12` *and* `auipc ra, imm20; jalr ra, ra, imm12`

Conditional Branches

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

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

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

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

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

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

The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC and Xtensa ISA), 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.

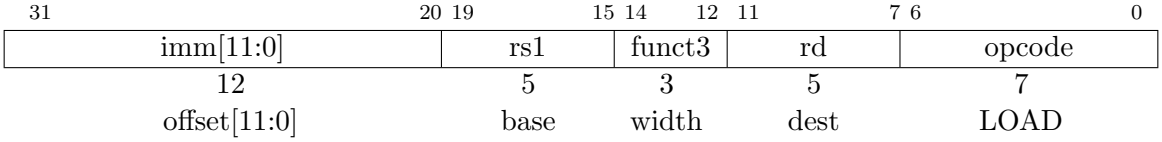
We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs.

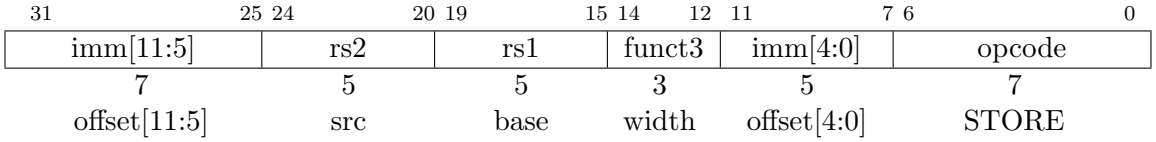
We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point 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 [13, 17, 16] and have been implemented in commercial processors [27]. 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 [27].

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 user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access. Loads with a destination of x0 must still raise any exceptions and action any other side effects even though the load value is discarded.





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 byte 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.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

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

We do not mandate atomicity for misaligned accesses so simple implementations can just use a machine trap and 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.

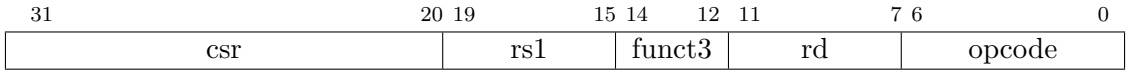
2.7 Control and Status Register Instructions

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 this section, with the two other user-level SYSTEM instructions 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.

CSR Instructions

We define the full set of CSR instructions here, although in the standard user-level base ISA, only a handful of read-only counter CSRs are accessible.



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 unaffected (though CSRs might have side effects when 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 unaffected.

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, such as raising

illegal instruction exceptions on accesses to read-only CSRs. 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.

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. 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.

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 a CSR, the update occurs after the execution of the instruction. In particular, a value written to `instret` by one instruction will be the value read by the following instruction (i.e., the increment of `instret` caused by the first instruction retiring happens before the write of the new value).

The assembler pseudo-instruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudo-instruction 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 pseudo-instructions 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*.

Timers and Counters

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

RV32I provides a number of 64-bit read-only user-level counters, which are mapped into the 12-bit CSR address space and accessed in 32-bit pieces using CSRRS instructions.

The RDCYCLE pseudo-instruction 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.

The RDTIME pseudo-instruction 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 in-

struction 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.

The RDINSTRET pseudo-instruction 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 that 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 between reading its upper and lower halves.

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

Figure 2.5: Sample code for reading the 64-bit cycle counter in RV32.

We mandate these basic counters be provided in all 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.

2.8 Environment Call and Breakpoints

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

The ECALL instruction is used to make a request to the supporting execution environment, which is

usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

The EBREAK instruction is used by debuggers to cause control to be transferred back 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.

2.9 Memory Consistency Model

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

The RISC-V ISA also provides the optional Ztso extension which imposes the stronger RVTSO (RISC-V Total Store Ordering) memory model for hardware that chooses to provide it. RVTSO provides a simpler programming model but is more restrictive on the implementations that can be legally built.

Under both models, code running on a single hart will appear to execute in order from the perspective of other memory operations 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, under both models, multithreaded code may require explicit synchronization to guarantee ordering between memory operations from different harts. The base RISC-V ISA provides a `fence` instruction for this purpose, described in Section A.3.4, while the optional "A" atomics extension defines load-reserved/store-conditional and atomic read-modify-write operations.

Appendix A provides a formal model and additional explanatory material about the memory model.

The RVWMO and RVTSO models are formalized as presented, but the interaction of the memory model with I/O, instruction fetches, page table walks, and `sfence.vma` *is not formalized. We may formalize some or all of them in a future revision. The "V" vector, transactional memory, and "J" JIT extensions will need to be incorporated into a future revision as well.*

Memory models which permit memory accesses of different sizes are an active and open area of research, and the specifics of how the RISC-V memory model deals with mixed-size memory accesses is subject to change. Nevertheless, we provide an educated guess at sane rules for such situations as a guideline for architects and implementers.

2.9.1 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 accesses produced by all harts. In general, a multithreaded program will have many

different possible executions, and each execution will have its its own corresponding global memory order.

The global memory order is defined in terms of the primitive load(s) and/or store(s) generated by each memory instruction. It is then subject to the constraints defined in the rest of this chapter. Any execution which satisfies all of the memory model constraints is a legal execution (as far as the memory model is concerned).

Memory Model Primitives

The RVWMO memory model is specified in terms of the memory accesses generated by each instruction. The *program order* over memory accesses reflects the order in which the instructions that generate each load and store are originally 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. Table 2.2 summarizes the memory accesses generated by each type of memory instruction.

RISC-V Instruction	Memory Accesses
`l{b|h|w|d}`	load*
`s{b|h|w|d}`	store*
`lr`	load
`lr.aq`	load-acquire-RCpc
`lr.aqrl`	load-acquire-RCsc
`lr.rl`	(deprecated)
`sc`	store
`sc.rl`	store-release-RCpc
`sc.aqrl`	store-release-RCsc
`sc.aq`	(deprecated)
`amo<op>`	load; <*op*>; store
`amo<op>.aq`	load-acquire-RCpc; <*op*>; store
`amo<op>.rl`	load; <*op*>; store-release-RCpc
`amo<op>.aqrl`	load-SC; <*op*>; store-SC

*: possibly multiple if misaligned

Table 2.2: Mapping of instructions into memory accesses, for the purposes of describing the memory model. "`.sc`" is a synonym for "`.aqrl`".

Every aligned load or store instruction gives rise to exactly one memory access that executes in a single-copy atomic fashion: it can never be observed in a partially-incomplete state. Every misaligned load or store may be decomposed into a set of component loads or stores at any granularity. The memory accesses generated by misaligned loads and stores are not ordered with respect to each other in program order, but they are ordered with respect to the memory accesses generated by preceding and subsequent instructions in program order.

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

AMOs give rise to exactly two memory accesses: one load and one store. These accesses are said to be *paired*. Every `lr` instruction gives rise to exactly one load. Every `sc` instruction gives rise to either zero stores or one store, depending on whether the store conditional succeeds. An `lr` instruction is said to be paired with the first `sc` instruction that follows it in program order, unless the `sc` is not successful or there is another `lr` instruction in between. The memory accesses generated by paired `lr` and (successful) `sc` instructions are also said to be paired. Both `lr` and `sc` instructions may be unpaired if they do not meet the conditions above. The complete list of conditions determining the success or failure of store conditional instructions is defined in the "A" extension.

Loads and stores generated by atomics may carry ordering annotations. Loads may carry "acquire-RCpc" or "acquire-RCsc" annotations. The term "load-acquire" without further annotation refers to both collectively. Stores may carry "release-RCpc" or "release-RCsc" annotations, and once again "store-release" without further annotation refers to both together. 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. Finally, AMOs with both `.aq` and `.rl` set are sequentially-consistent in an even stronger sense: they do not allow any reordering in either direction. The precise semantics of these annotations as they apply to RVWMO are described by the memory model rules below.

Although the ISA does not currently contain `l{b|h|w|d}.aq[rl]` *or* `s{b|h|w|d}.[aq]rl` *instructions, we may add them as assembler pseudoinstructions to facilitate forwards compatibility with their potential future addition into the ISA. These pseudoinstructions will generally assemble per the fence-based mappings of Section A.6 until if and when the instructions are added to the ISA. The RVWMO memory model is also designed to easily accommodate the possible future inclusion of such instructions.*

Dependencies

The definition of the RVWMO memory model depends in part on the notion of a syntactic dependency. A register r read by an instruction b has a syntactic dependency on an instruction a if a precedes b in program order, r is not `x0`, and either of the following hold:

1. r is written by a and read by b, and no other instruction between a and b in program order modifies r
2. There is some other instruction i such that a register read by i has a dependency on a, and a register read by b has a dependency on i

Specific types of dependencies are defined as follows. First, for two instructions a and b, b has a syntactic address dependency on a if a register used to calculate the address accessed by b has a syntactic dependency on a. Second, b has a syntactic data dependency on a if b is a store and a register used to calculate the data being written by b has a syntactic dependency on a. Third, b has a syntactic control dependency on a if there exists a branch m between a and b in program order such that a register checked as part of the condition of m has a syntactic dependency on a. Finally, b has a syntactic success dependency on a if a is a store-conditional and a register read by b has a register dependency on the store conditional success register written by a.

Preserved Program Order

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

The complete definition of preserved program order is as follows: memory access a precedes memory access b in preserved program order (and hence also in the global memory order) if a precedes b in program order, a and b are both accesses to normal memory (i.e., not to I/O regions), and any of the following hold:

- Basic same-address orderings:
  1. a and b are accesses to overlapping memory addresses, and b is a store
- Explicit synchronization
  2. a and b are separated in program order by a fence, a is in the predecessor set of the fence, and b is in the successor set of the fence
  3. a is a load-acquire
  4. b is a store-release
  5. a is a store-release-RCsc and b is a load-acquire-RCsc
  6. a is a store-SC
  7. b is a load-SC
- Dependencies
  8. a is a load, and b has a syntactic address dependency on a
  9. a is a load, b is a store, and b has a syntactic data dependency on a
  10. a is a load, b is a store, and b has a syntactic control dependency on a
  11. a is a load, and there exists some m such that m has an address or data dependency on a and b has a success dependency on m
- Same-address load-load ordering
  12. a and b are loads to overlapping memory addresses, there is no store to overlapping memory location(s) between a and b in program order, and a and b return values from different stores
- Pipeline dependency artifacts
  13. a and b are loads, and there exists some store m between a and b in program order such that m has an address or data dependency on a, m accesses a memory address that overlaps the address accessed by b, and there is no other store to an overlapping memory location(s) between m and b in program order
  14. 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. a and b are loads, b has a syntactic control dependency on a, and there exists a `fence.i` between the branch used to form the control dependency and b in program order
16. a is a load, there exists an instruction m which has a syntactic address dependency on a, and there exists a `fence.i` between m and b in program order

Memory Model Axioms

An execution of a RISC-V program obeys the RVWMO memory consistency model only if it obeys the *load value axiom* and the *atomicity axiom*.

Load Value Axiom: Each byte of each load returns the corresponding byte written by the whichever of the following two stores comes later in the global memory order:

1. the latest store to the same address and preceding the load in the global memory order
2. the latest store to the same address and preceding the load in program order

Atomicity Axiom: If r and w are a paired load and store, and if s is any store from which r returns a value, then there can be no store from another hart to an overlapping memory location which follows both r and s and which precedes w in the global memory order.

2.9.2 Definition of the RVTSO Memory Model

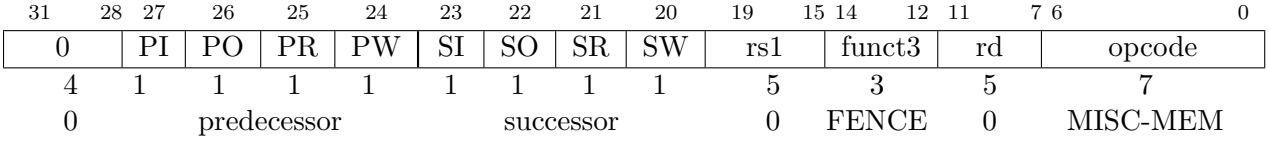
RISC-V cores which implement Ztso impose RVTSO onto all memory accesses. RVTSO behaves just like RVWMO but with the following modifications:

- All `l{b|h|w|d|r}` instructions behave as if `.aq` is set
- All `s{b|h|w|d|c}` instructions behave as if `.rl` is set
- All AMO instructions behave as if `.aq` and `.rl` are both set

These rules render PPO rules 1 and 8–16 redundant. They also make redundant any non-I/O fences that do not have both `.pw` and `.sr` set. Finally, they also imply that all AMO instructions are fully-fenced; nothing will be reordered past an AMO.

The definitions of RVTSO and Ztso are new and have not yet been reviewed as carefully as the RVWMO model. For example, it is not yet properly specified how LR/SC will behave under RVTSO.

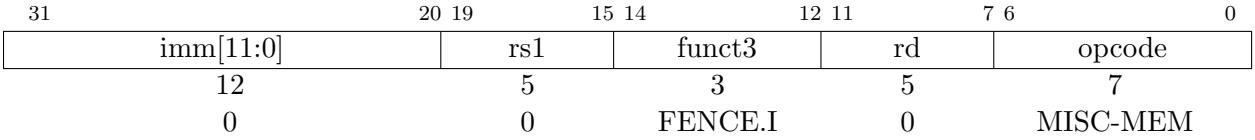
2.10 Memory Ordering Instructions



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. The execution environment will define what I/O operations are possible, and in particular, which load and store instructions might 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 coprocessor I/O instructions that will also be ordered using the I and O bits in a FENCE.

The unused fields in the FENCE instruction, *imm[11:8]*, *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.

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.*



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 the same RISC-V hart until a FENCE.I instruction is executed. A FENCE.I instruction only 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 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 ABI will provide mechanisms for multiprocessor instruction-stream synchronization.

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, then only the pipeline needs to be flushed at a FENCE.I.

We considered but did not include a "store instruction word" instruction (as in MAJC [30]). 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.

Chapter 3

RV32E Base Integer Instruction Set, Version 1.9

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

RV32E was designed to provide an even smaller base core for embedded microcontrollers. Although we had mentioned this possibility in version 2.0 of this document, we initially resisted defining this subset. However, given the demand for the smallest possible 32-bit microcontroller, and in the interests of preempting fragmentation in this space, we have now defined RV32E as a fourth standard base ISA in addition to RV32I, RV64I, and RV128I. The E variant is only standardized for the 32-bit address space width.

3.1 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. Systems with hardware floating-point must use an I base.

3.2 RV32E Instruction Set

RV32E uses the same instruction set encoding as RV32I, except that use of register specifiers `x16`–`x31` in an instruction will result in an illegal instruction exception being raised.

Any future standard extensions will not make use of the instruction bits freed up by the reduced register-specifier fields and so these are available for non-standard extensions.

A further simplification is that the counter instructions (`rdcycle[h]`,`rdtime[h]`, `rdinstret[h]`) are no longer mandatory.

The mandatory counters require additional registers and logic, and can be replaced with more application-specific facilities.

3.3 RV32E Extensions

RV32E can be extended with the M, A, and C user-level standard extensions.

We do not intend to support hardware floating-point with the RV32E subset. The savings from reduced register count become negligible in the context of a hardware floating-point unit, and we wish to reduce the proliferation of ABIs.

The privileged architecture of an RV32E system can include user mode as well as machine mode, and the physical memory protection scheme described in Volume II.

We do not intend to support full Unix-style operating systems with the RV32E subset. The savings from reduced register count become negligible in the context of an OS-capable core, and we wish to avoid OS fragmentation.

Chapter 4

RV64I Base Integer Instruction Set, Version 2.0

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

4.1 Register State

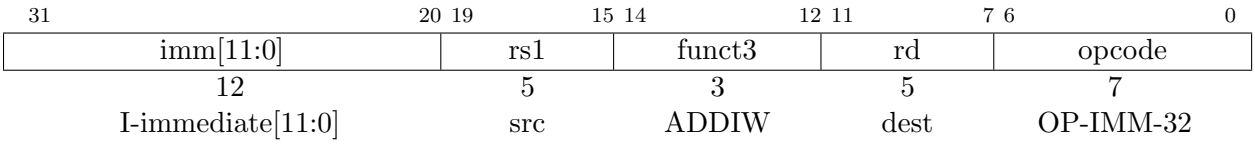
RV64I widens the integer registers and supported user address space to 64 bits (XLEN=64 in Figure 2.1).

4.2 Integer Computational Instructions

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, i.e. bits XLEN-1 through 31 are equal. They cause an illegal instruction exception in RV32I.

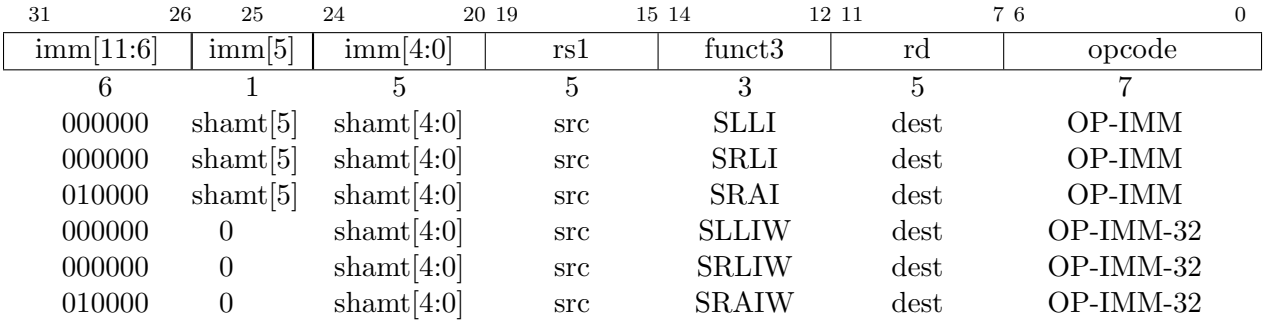
The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion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

Integer Register-Immediate Instructions



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

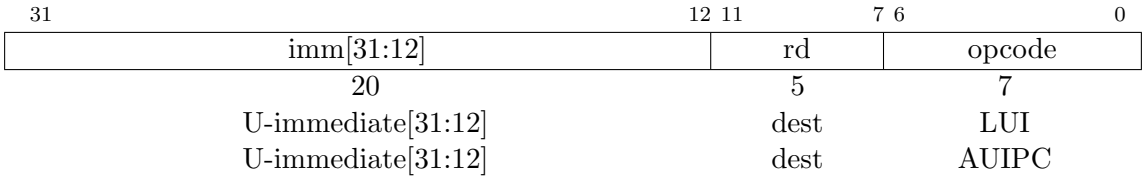
ADDIW is an RV64I-only instruction that adds the sign-extended 12-bit immediate to register *rs1* and produces the proper sign-extension of a 32-bit result in *rd*. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW *rd, rs1, 0* writes the sign-extension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



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

Shifts by a constant are encoded as a specialization of the I-type format using the same instruction opcode as RV32I. The operand to be shifted is in *rs1*, and the shift amount is encoded in the lower 6 bits of the I-immediate field for RV64I. The right shift type is encoded in bit 30. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). For RV32I, SLLI, SRLI, and SRAI generate an illegal instruction exception if $imm[5] \neq 0$.

SLLIW, SRLIW, and SRAIW are RV64I-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. SLLIW, SRLIW, and SRAIW generate an illegal instruction exception if $imm[5] \neq 0$.



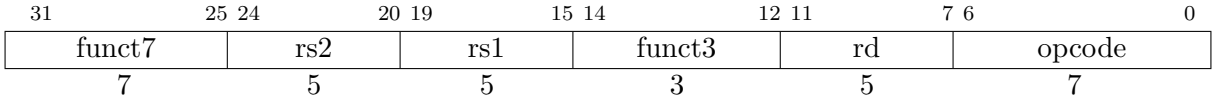
31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

LUI (load upper immediate) uses the same opcode as RV32I. LUI places the 20-bit U-immediate into bits 31–12 of register *rd* and places zero in the lowest 12 bits. The 32-bit result is sign-extended to 64 bits.

AUIPC (add upper immediate to `pc`) uses the same opcode as RV32I. AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC appends 12 low-

order zero bits to the 20-bit U-immediate, sign-extends the result to 64 bits, then adds it to the `pc` and places the result in register *rd*.

Integer Register-Register Operations



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

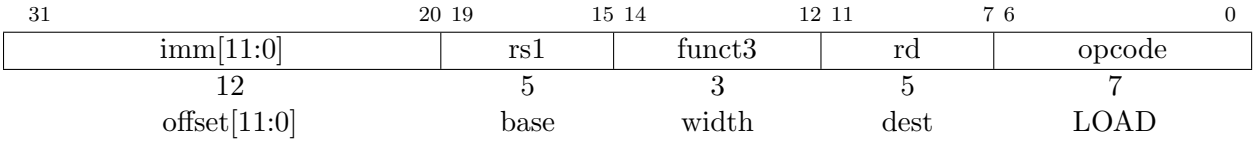
ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

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

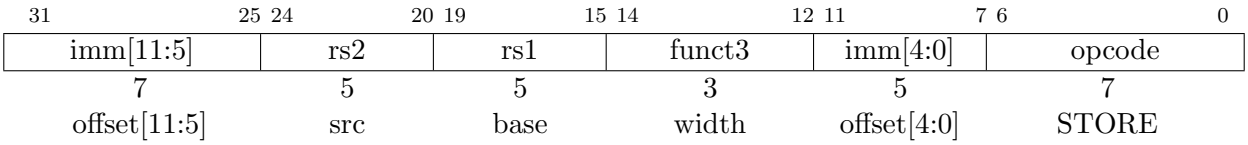
SLLW, SRLW, and SRAW are RV64I-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. The shift amount is given by *rs2[4:0]*.

4.3 Load and Store Instructions

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



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



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

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

The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register *rd* for RV64I. The LWU instruction, on the other hand, zero-extends the 32-bit value

from memory for RV64I. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register *rs2* to memory respectively.

4.4 System Instructions

In RV64I, the CSR instructions can manipulate 64-bit CSRs. In particular, the RDCYCLE, RDTIME, and RDINSTRET pseudo-instructions read the full 64 bits of the `cycle`, `time`, and `instret` counters. Hence, the RDCYCLEH, RDTIMEH, and RDINSTRETH instructions are not necessary and are illegal in RV64I.

Chapter 5

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 1 PB 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 100 PB 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, and a new set of "*D" integer instructions that operate on 64-bit values held in the low bits of the 128-bit integer registers are added. The "*D" instructions consume two major opcodes (OP-IMM-64 and OP-64) in the standard 32-bit encoding.

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.

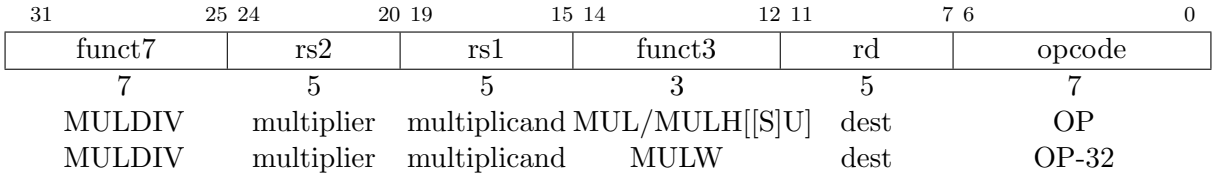
Chapter 6

"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.

6.1 Multiplication Operations

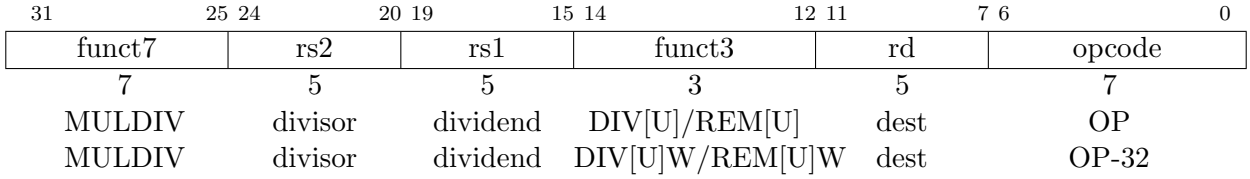


MUL performs an XLEN-bit×XLEN-bit multiplication and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and signed×unsigned multiplication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

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

MULW is only valid for RV64, and 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. 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.

6.2 Division Operations



DIV and DIVU perform signed and unsigned integer division of XLEN bits by XLEN bits, rounding towards zero. REM and REMU provide the remainder of the corresponding division operation. 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.

The semantics for division by zero and division overflow are summarized in Table 6.1. The quotient of division by zero has all bits set, i.e. $2^{XLEN} - 1$ for unsigned division or -1 for signed division. The remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer, -2^{XLEN-1}, is divided by -1. The quotient of signed division overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

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

Table 6.1: Semantics for division by zero and division overflow.

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.

DIVW and DIVUW instructions are only valid for RV64, and 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 instructions are only valid for RV64, and 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.

Chapter 7

"A" Standard Extension for Atomic Instructions, Version 2.0

This section is somewhat out of date as the RISC-V memory model is currently under revision to ensure it can efficiently support current programming language memory models. The revised base memory model will contain further ordering constraints, including at least that loads to the same address from the same hart cannot be reordered, and that syntactic data dependencies between instructions are respected.

The standard atomic instruction extension is denoted by instruction subset name "A", and 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 [10].

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.

7.1 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 [10], 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 can not 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 can only be observed by any other hart in the same global order of all sequentially consistent atomic memory operations to the same address domain.

Theoretically, the definition of the aq *and* rl *bits allows for implementations without global store atomicity. When both* aq *and* rl *bits are set, however, we require full sequential consistency for the atomic operation which implies global store atomicity in addition to both acquire and release semantics. In practice, hardware systems are usually implemented with global store atomicity, embodied in local processor ordering rules together with single-writer cache coherence protocols.*

7.2 Load-Reserved/Store-Conditional Instructions

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

Complex atomic memory operations on a single memory word are performed with the load-reserved (LR) and store-conditional (SC) instructions. LR loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a reservation on the memory address. SC writes a word in *rs2* to the address in *rs1*, provided a valid reservation still exists on that address. SC writes zero to *rd* on success or a nonzero code on failure.

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

The main disadvantage of LR/SC over CAS is livelock, which we avoid 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.

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.

The execution environment may require 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, in such an execution environment, an LR or SC address is not naturally aligned, a misaligned address exception will be generated.

If the execution environment permits misaligned LR and SC addresses, then LR and SC instructions using misaligned addresses execute atomically with respect to other accesses to the same address and of the same size. In such environments, regular loads and stores using misaligned addresses also execute atomically with respect to other accesses to the same address and of the same size.

In the standard A extension, certain constrained LR/SC sequences are guaranteed to succeed eventually. The static code for the LR/SC sequence plus the code to retry the sequence in case of failure must comprise at most 16 integer instructions placed sequentially in memory. For the sequence to be guaranteed to eventually succeed, the dynamic code executed between the LR and SC instructions can only contain other instructions from the base "I" subset, excluding loads, stores, backward jumps or taken backward branches, FENCE, FENCE.I, and SYSTEM instructions. The code to retry a failing LR/SC sequence can contain backward jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraints. The SC must be to the same address and of the same data size as the latest LR executed. LR/SC sequences that do not meet these constraints might complete on some attempts on some implementations, but there is no guarantee of eventual success.

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 forward progress to LR/SC atomic sequences. The restrictions on LR/SC sequence contents allows an implementation to capture a cache line on the LR and complete the LR/SC sequence by holding off remote cache interventions for a bounded short time. Interrupts and TLB misses might cause the reservation to be lost, but eventually the atomic sequence can complete. We restricted the length of LR/SC sequences 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 sequences to avoid restrictions on data cache associativity. The restrictions on branches and jumps limits 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.

An implementation can reserve an arbitrary subset of the memory space on each LR and multiple LR reservations might be active simultaneously for a single hart. An SC can succeed if no accesses from other harts to the address can be observed to have occurred between the SC and the last LR in this hart to reserve the address. Note this LR might have had a different address argument, but

reserved the SC's address as part of the memory subset. 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. The SC must fail if there is an observable memory access from another hart to the address, or a preemptive context switch on this hart, between the LR and the SC.

The privileged architecture specifies the mechanism by which the implementation yields a load reservation during a preemptive context switch.

Cooperative user-level context switches might not cause a load reservation to be yielded, so user-level threads should generally avoid voluntary context switches in the middle of an LR/SC sequence.

The specification explicitly allows implementations to support more powerful implementations with wider guarantees, provided they do not void the atomicity guarantees for the constrained sequences.

LR/SC can be used to construct lock-free data structures. An example using LR/SC to implement a compare-and-swap function is shown in Figure A.20. If inlined, compare-and-swap functionality need only take three instructions.

```
    # 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 a0, a2, (a0)    # Try to update.
    jr ra                # Return.
fail:
    li a0, 1             # Set return to failure.
    jr ra                # Return.
```

Figure 7.1: Sample code for compare-and-swap function using LR/SC.

An SC instruction can never be observed by another RISC-V hart before the immediately preceding LR. Due to the atomic nature of the LR/SC sequence, no memory operations from any hart can be observed to have occurred between the LR and a successful SC. The LR/SC sequence can be given acquire semantics by setting the *aq* bit on the SC instruction. The LR/SC sequence can be given release semantics by setting the *rl* bit on the LR instruction. Setting both *aq* and *rl* bits on the LR instruction, and setting the *aq* bit on the SC instruction makes the LR/SC sequence sequentially consistent with respect to other sequentially consistent atomic operations.

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.

In general, a multi-word atomic primitive is desirable but there is still considerable debate about what form this should take, and guaranteeing forward progress adds complexity to a system. Our current thoughts are to include a small limited-capacity transactional memory buffer along the lines of the original transactional memory proposals as an optional standard extension "T".

7.3 Atomic Memory Operations

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

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

The execution environment may require 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, in such an execution environment, an AMO address is not naturally aligned, a misaligned address exception will be generated.

If the execution environment permits misaligned AMO addresses, then AMO instructions using misaligned addresses execute atomically with respect to other accesses to the same address and of the same size. In such environments, regular loads and stores using misaligned addresses also execute atomically with respect to other accesses to the same address and of the same size.

The operations supported are swap, integer add, logical AND, logical OR, logical 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. 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.

The AMOs were designed to implement the C11 and C++11 memory models efficiently. Al-

though 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-set spinlock is shown in Figure A.21. 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:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez         t0, again    # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Figure 7.2: Sample code for mutual exclusion. `a0` contains the address of the lock.

We recommend the use of the AMO Swap idiom shown above for both lock acquire and release to simplify the implementation of speculative lock elision [25].

At the risk of complicating the implementation of atomic operations, microarchitectures can elide the store within the acquire swap if the lock value matches the swap value, to avoid dirtying a cache line held in a shared or exclusive clean state. The effect is similar to a test-and-test-and-set lock but with shorter code paths.

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 8

"F" Standard Extension for Single-Precision Floating-Point, Version 2.0

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 [14].

8.1 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 8.1. We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, and FLEN=32 for the F single-precision floating-point extension. Most floating-point instructions operate on values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.

We considered a unified register file for both integer and floating-point values as this simplifies software register allocation and calling conventions, and reduces total user state. However, a split organization increases the total number of registers accessible with a given instruction width, simplifies provision of enough regfile ports for wide superscalar issue, supports decoupled floating-point-unit architectures, and simplifies use of internal floating-point encoding techniques. Compiler support and calling conventions for split register file architectures are well understood, and using dirty bits on floating-point register file state can reduce context-switch overhead.

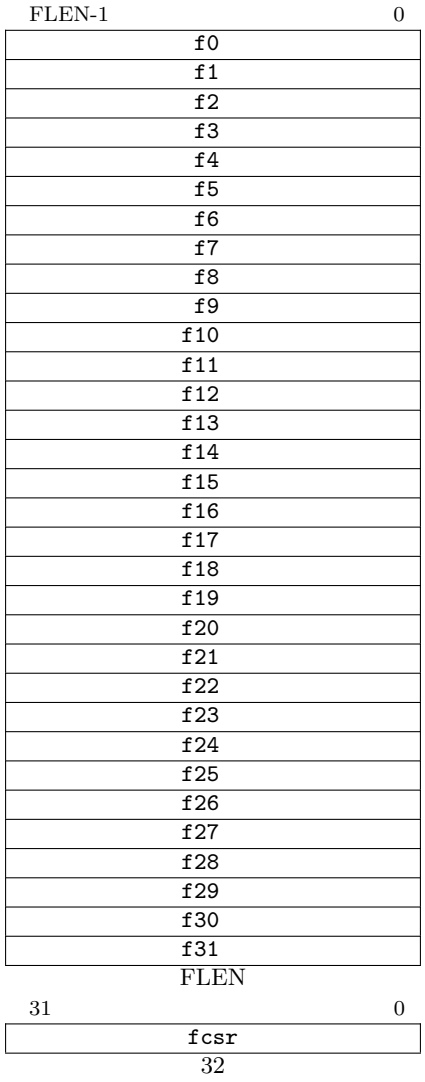


Figure 8.1: RISC-V standard F extension single-precision floating-point state.

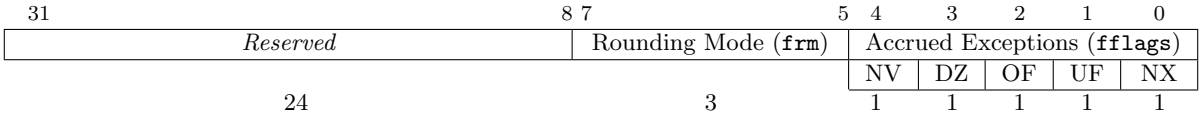


Figure 8.2: Floating-point control and status register.

8.2 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 8.2.

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudo-ops 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 pseudo-ops are defined for these accesses. The FRRM instruction reads the Rounding Mode field `frm` and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in `frm` by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into `frm`. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field `fflags`.

Bits 31–8 of the `fcsr` are reserved for other standard extensions, including the "L" standard extension for decimal floating-point. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in Table 8.1. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will cause an illegal instruction trap. Some instructions that have the *rm* field are nevertheless unaffected by the rounding mode; they should have their *rm* field set to RNE (000).

The C99 language standard effectively mandates the provision of a dynamic rounding mode register.

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

Table 8.1: Rounding mode encoding.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 8.2.

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

Table 8.2: Accrued exception flag encoding.

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

8.3 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.

8.4 Subnormal Arithmetic

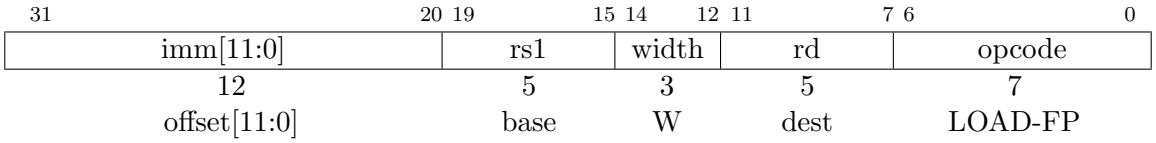
Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

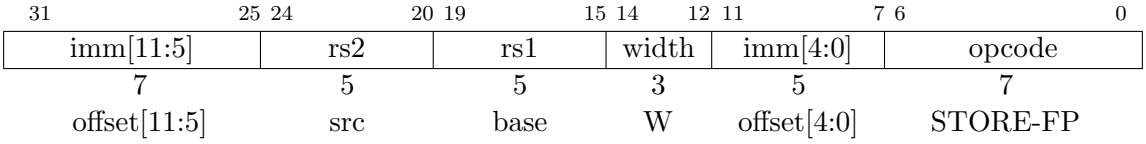
In the parlance of the IEEE standard, tininess is detected after rounding.

Detecting tininess after rounding results in fewer spurious underflow signals.

8.5 Single-Precision Load and Store Instructions

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





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

8.6 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, FSUB.S, FMUL.S, and FDIV.S perform single-precision floating-point addition, subtraction, multiplication, and division, respectively, between *rs1* and *rs2*, writing the result to *rd*. FSQRT.S computes the square root of *rs1* and writes the result to *rd*.

The 2-bit floating-point format field *fmt* is encoded as shown in Table 8.3. It is set to *S* (00) for all instructions in the F extension.

All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in Table 8.1.

Floating-point minimum-number and maximum-number instructions FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. For the purposes of these instructions only,

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

Table 8.3: Format field encoding.

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 raise the invalid operation exception, even when the result is not NaN.

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

Floating-point fused multiply-add instructions require a new standard instruction format. R4-type instructions specify three source registers (*rs1*, *rs2*, and *rs3*) and a destination register (*rd*). This format is only used by the floating-point fused multiply-add instructions. Fused multiply-add instructions multiply the values in *rs1* and *rs2*, optionally negate the product, then add or subtract the value in *rs3*, writing the final result to *rd*. FMADD.S computes *rs1*×*rs2*+*rs3*; FMSUB.S computes *rs1*×*rs2*-*rs3*; FNMSUB.S computes -*rs1*×*rs2*+*rs3*; and FNMADD.S computes -*rs1*×*rs2*-*rs3*.

The fused multiply-add instructions must raise the invalid operation exception when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

The IEEE 754-2008 standard permits, but does not require, raising the invalid exception for the operation $\infty \times 0 +$ *qNaN.*

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

8.7 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 RV64, FCVT.W[U].S sign-extends the 32-bit result. FCVT.L[U].S and FCVT.S.L[U] are illegal in RV32. 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 8.4 gives the range of valid inputs for FCVT.*int*.S and the 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	$2^{31}-1$	$2^{32}-1$	$2^{63}-1$	$2^{64}-1$

Table 8.4: Domains of float-to-integer conversions and behavior for invalid inputs.

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

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

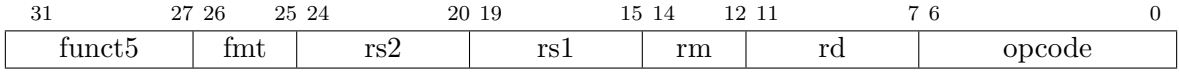
Floating-point to floating-point sign-injection instructions, FSGNJ.S, FSGNJN.S, and FSGNJX.S, produce a result that takes all bits except the sign bit from *rs1*. For FSGNJ, the result's sign bit is *rs2*'s sign bit; for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit; and for FSGNJX,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

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

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

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.W moves the single-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the lower 32 bits of integer register *rd*. 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.



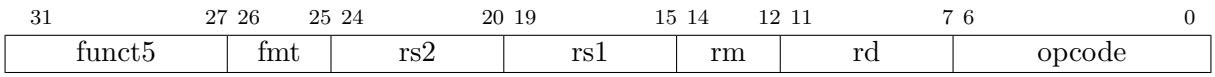
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

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

8.8 Single-Precision Floating-Point Compare Instructions

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

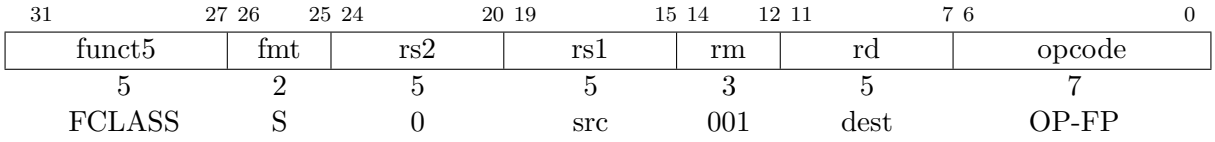
FLT.S and FLE.S perform what the IEEE 754-2008 standard refers to as *signaling* comparisons: that is, an Invalid Operation exception is raised if either input is NaN. FEQ.S performs a *quiet* comparison: only signaling NaN inputs cause an Invalid Operation exception. For all three instructions, the result is 0 if either operand is NaN.



funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	S	src2	src1	EQ/LT/LE	dest	OP-FP

8.9 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 8.5. The corresponding bit in *rd* will be set if the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in *rd* will be set. FCLASS.S does not set the floating-point exception flags.



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

Table 8.5: Format of result of FCLASS instruction.

Chapter 9

"D" Standard Extension for Double-Precision Floating-Point, Version 2.0

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.

9.1 D Register State

The D extension widens the 32 floating-point registers, `f0`–`f31`, to 64 bits (FLEN=64 in Figure 8.1). The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section 9.2.

FLEN can be 32, 64, or 128 depending on which of the F, D, and Q extensions are supported. There can be up to four different floating-point precisions supported, including H, F, D, and Q. Half-precision H scalar values are only supported if the V vector extension is supported.

9.2 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.

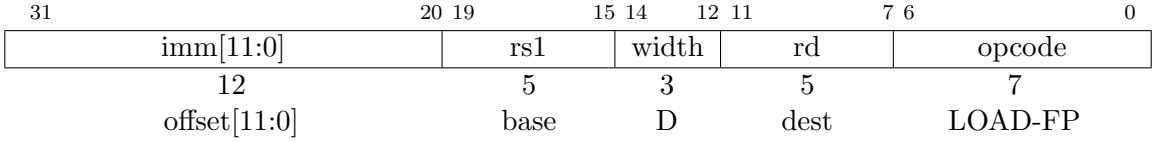
Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in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

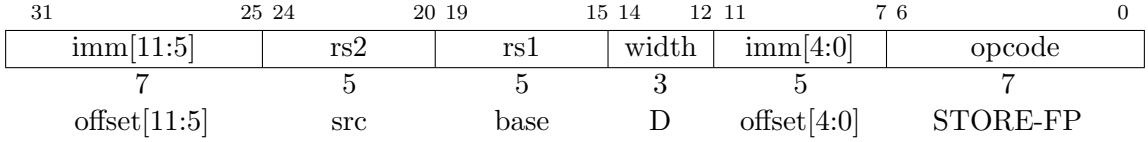
Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can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

9.3 Double-Precision Load and Store Instructions

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

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

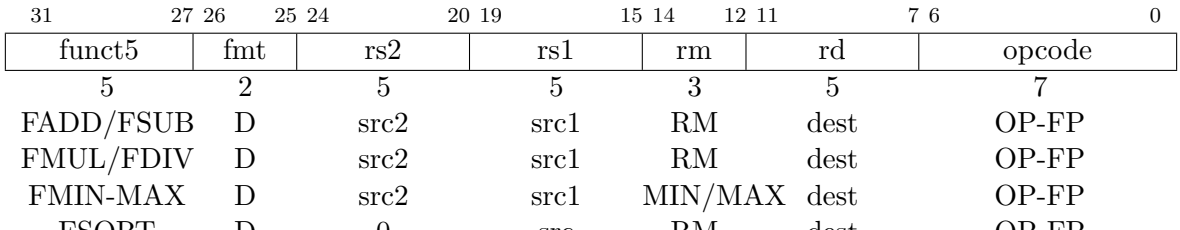


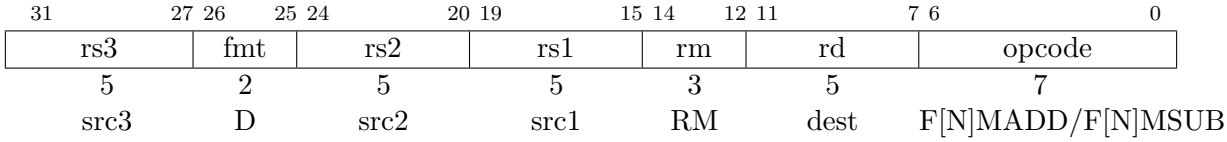


FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64.

9.4 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.





9.5 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 illegal in RV32. The range of valid inputs for FCVT.*int*.D and the behavior for invalid inputs are the same as for FCVT.*int*.S.

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

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

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

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

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

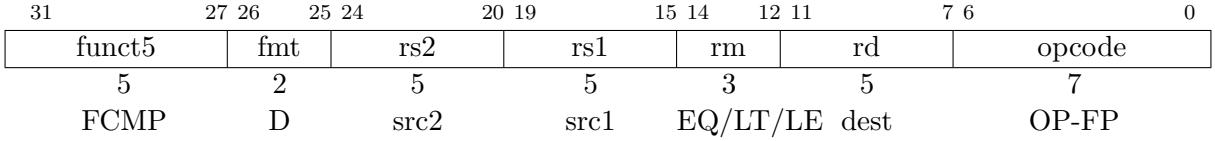
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

For RV64 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*.

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

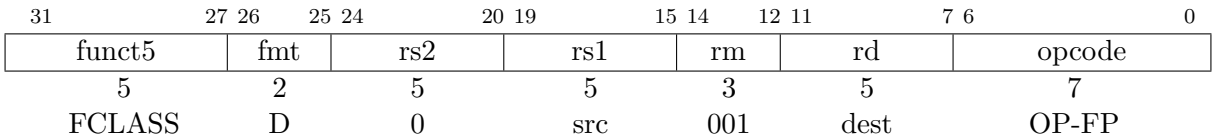
9.6 Double-Precision Floating-Point Compare Instructions

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



9.7 Double-Precision Floating-Point Classify Instruction

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



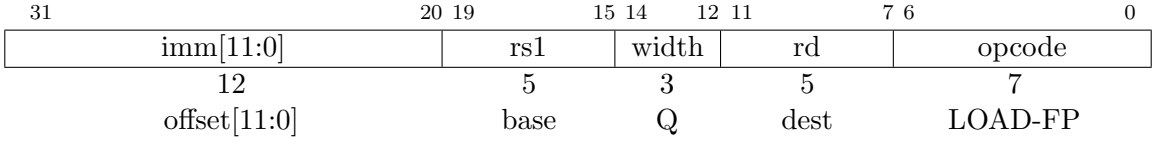
Chapter 10

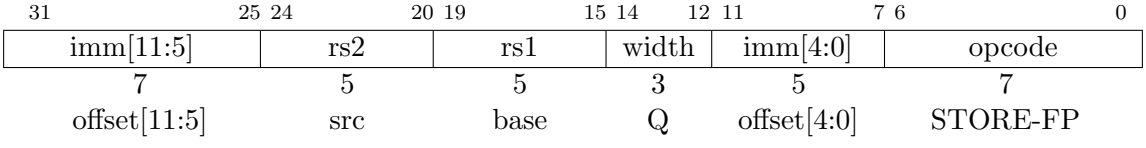
"Q" Standard Extension for Quad-Precision Floating-Point, Version 2.0

This chapter describes the Q standard extension for 128-bit binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The 128-bit or quad-precision binary floating-point instruction subset is named "Q", and requires RV64IFD. 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 9.2 is now extended recursively to allow a single-precision value to be NaN-boxed inside a double-precision value which is itself NaN-boxed inside a quad-precision value.

10.1 Quad-Precision Load and Store Instructions

New 128-bit variants of LOAD-FP and STORE-FP instructions are added, encoded with a new value for the funct3 width field.





FLQ and FSQ are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN=128.

10.2 Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 10.1.

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

Table 10.1: Format field encoding.

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

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

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

10.3 Quad-Precision Convert and Move Instructions

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

New floating-point to floating-point conversion instructions FCVT.S.Q, FCVT.Q.S, FCVT.D.Q, FCVT.Q.D are added.

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

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

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

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

RV128 supports FMV.X.Q and FMV.Q.X in the Q extension.

10.4 Quad-Precision Floating-Point Compare Instructions

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

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

10.5 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.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	Q	0	src	001	dest	OP-FP

Chapter 11

"L" Standard Extension for Decimal Floating-Point, Version 0.0

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

11.1 Decimal Floating-Point Registers

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

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

Chapter 12

"C" Standard Extension for Compressed Instructions, Version 2.0

This chapter describes the current draft 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.

12.1 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. With the addition of the C extension, JAL and JALR instructions will no longer raise an instruction misaligned exception.

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

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

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

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

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

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

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

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

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

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch [6], 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 [3] 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 [28], 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 [33].*

12.2 Compressed Instruction Formats

Table 12.1 shows the eight compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, and CB are limited to just 8 of them. Table 12.2 lists these popular registers, which correspond to registers `x8` to `x15`. Note that there is a separate version of

load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow access to all 32 data registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.

The RISC-V ABI was changed to make the frequently used registers map to registers `x8`–`x15`*. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E subset base specification, which only has 16 integer registers.*

Compressed register-based floating-point loads and stores also use the CL and CS formats respectively, with the eight registers mapping to `f8` to `f15`.

The standard RISC-V calling convention maps the most frequently used floating-point registers to registers `f8` *to* `f15`*, which allows the same register decompression decoding as for integer register numbers.*

The formats were designed to keep bits for the two register source specifiers in the same place in all instructions, while the destination register field can move. When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding. Where immediates are sign-extended, the sign-extension is always from bit 12. Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate muxes required.

The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations. For example, immediate bits 17—10 are always sourced from the same instruction bit positions. Five other immediate bits (5, 4, 3, 1, and 0) have just two source instruction bits, while four (9, 7, 6, and 2) have three sources and one (8) has four sources.

For many RVC instructions, zero-valued immediates are disallowed and `x0` is not a valid 5-bit register specifier. These restrictions free up encoding space for other instructions requiring fewer operand bits.

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

Table 12.1: Compressed 16-bit RVC instruction formats.

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

Table 12.2: Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, and CB formats.

12.3 Load and Store Instructions

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

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

Stack-Pointer-Based Loads and Stores

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

These instructions use the CI format.

C.LWSP loads a 32-bit value from memory into register *rd*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `lw rd, offset[7:2](x2)`.

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[8:3](x2)`.

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[9:4](x2)`.

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[7:2](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[8:3](x2)`.

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

These instructions use the CSS format.

C.SWSP stores a 32-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `sw rs2, offset[7:2](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[8:3](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[9:4](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[7:2](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[8:3](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, 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 [34].

While reasonable architects might come to different conclusions, we decided to omit load and store multiple and instead use the software-only approach of calling save/restore millicode routines to attain the greatest code size reduction.

Register-Based Loads and Stores

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

These instructions use the CL format.

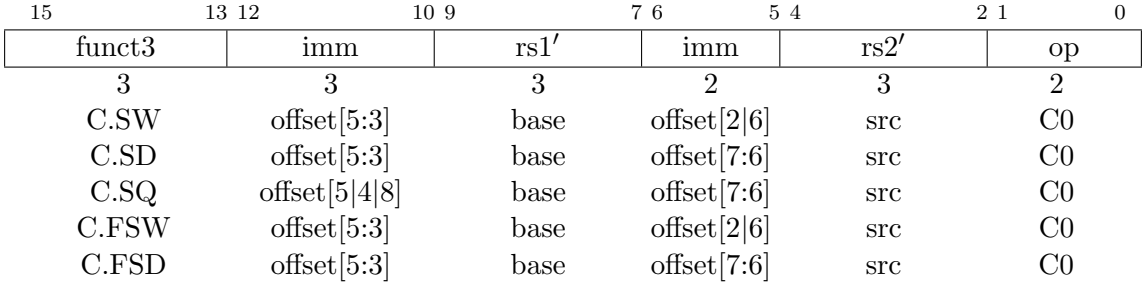
C.LW loads a 32-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `lw rd′, offset[6:2](rs1′)`.

C.LD is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `ld rd′, offset[7:3](rs1′)`.

C.LQ is an RV128C-only instruction that loads a 128-bit value from memory into register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1′*. It expands to `lq rd′, offset[8:4](rs1′)`.

C.FLW is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `flw rd′, offset[6:2](rs1′)`.

C.FLD is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd′*. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `fld rd′, offset[7:3](rs1′)`.



15–13	12–10	9–7	6–5	4–2	1–0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

These instructions use the CS format.

C.SW stores a 32-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `sw rs2′, offset[6:2](rs1′)`.

C.SD is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `sd rs2′, offset[7:3](rs1′)`.

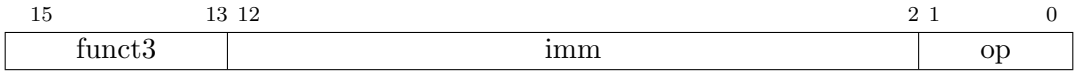
C.SQ is an RV128C-only instruction that stores a 128-bit value in register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the base address in register *rs1′*. It expands to `sq rs2′, offset[8:4](rs1′)`.

C.FSW is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the base address in register *rs1′*. It expands to `fsw rs2′, offset[6:2](rs1′)`.

C.FSD is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2′* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the base address in register *rs1′*. It expands to `fsd rs2′, offset[7:3](rs1′)`.

12.4 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.

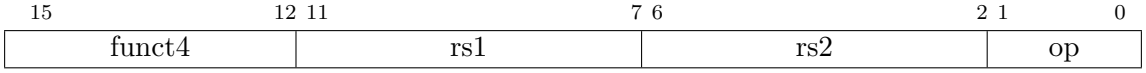


3	11	2
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1

These instructions use the CJ format.

C.J performs an unconditional control transfer. The offset is sign-extended and added to the `pc` to form the jump target address. C.J can therefore target a ±2 KiB range. C.J expands to `jal x0, offset[11:1]`.

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[11:1]`.



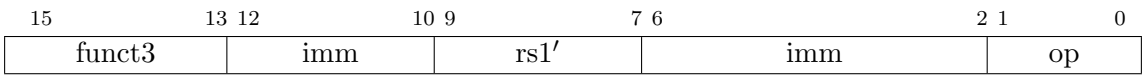
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

These instructions use the CR format.

C.JR (jump register) performs an unconditional control transfer to the address in register *rs1*. C.JR expands to `jalr x0, rs1, 0`.

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, rs1, 0`.

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.



3	3	3	5	2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

These instructions use the CB format.

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the `pc` to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register *rs1′* is zero. It expands to `beq rs1′, x0, offset[8:1]`.

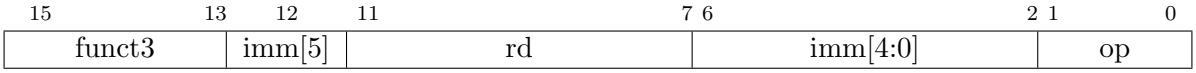
C.BNEZ is defined analogously, but it takes the branch if *rs1′* contains a nonzero value. It expands to `bne rs1′, x0, offset[8:1]`.

12.5 Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

Integer Constant-Generation Instructions

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



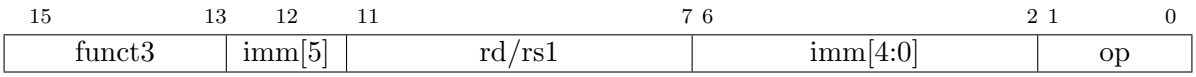
3	1	5	5	2
C.LI	imm[5]	dest≠0	imm[4:0]	C1
C.LUI	nzimm[17]	dest≠{0, 2}	nzimm[16:12]	C1

C.LI loads the sign-extended 6-bit immediate, *imm*, into register *rd*. C.LI is only valid when *rd*≠`x0`. C.LI expands into `addi rd, x0, imm[5:0]`.

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 is only valid when *rd*≠{`x0`, `x2`}, and when the immediate is not equal to zero. C.LUI expands into `lui rd, nzimm[17:12]`.

Integer Register-Immediate Operations

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



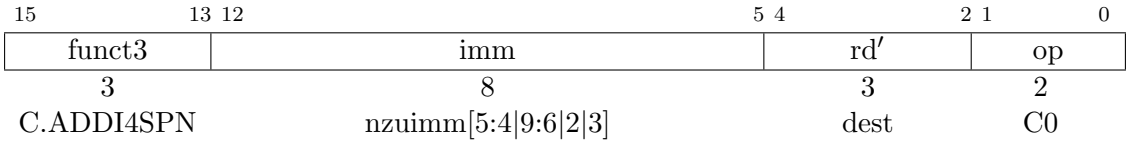
3	1	5	5	2
C.ADDI	nzimm[5]	dest≠0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register *rd* then writes the result to *rd*. C.ADDI expands into `addi rd, rd, nzimm[5:0]`. C.ADDI is only valid when *rd*≠`x0`.

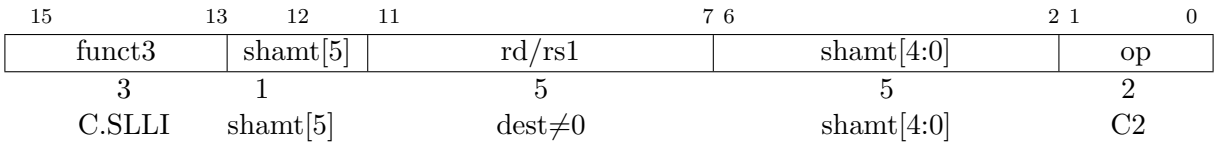
C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into `addiw rd, rd, imm[5:0]`. The immediate can be zero for C.ADDIW, where this corresponds to `sext.w rd`. C.ADDIW is only valid when *rd*≠`x0`.

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

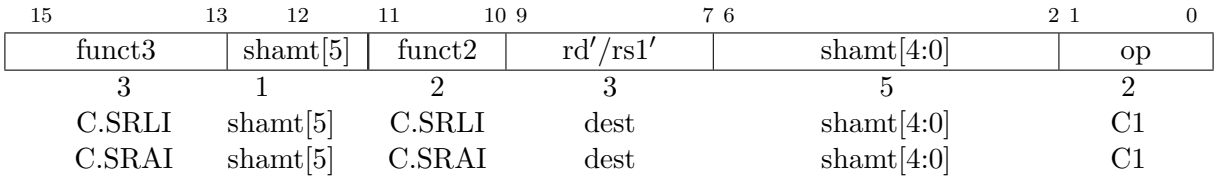
In the standard RISC-V calling convention, the stack pointer `sp` *is always 16-byte aligned.*



C.ADDI4SPN is a CIW-format RV32C/RV64C-only instruction that adds a *zero*-extended non-zero immediate, scaled by 4, to the stack pointer, `x2`, and writes the result to `rd′`. This instruction is used to generate pointers to stack-allocated variables, and expands to `addi rd′, x2, nzuimm[9:2]`.



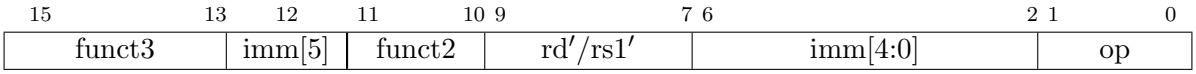
C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount of zero is used to encode a shift of 64. C.SLLI expands into `slli rd, rd, shamt[5:0]`, except for RV128C with `shamt=0`, which expands to `slli rd, rd, 64`.



C.SRLI is a CB-format instruction that performs a logical right shift of the value in register *rd′* then writes the result to *rd′*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. 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[5:0]`, except for RV128C with `shamt=0`, which expands to `srli rd′, rd′, 64`.

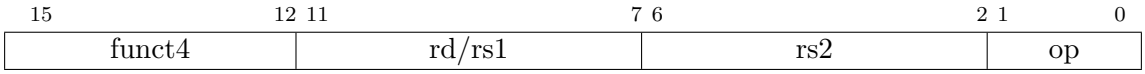
C.SRAI is defined analogously to C.SRLI, but instead performs an arithmetic right shift. C.SRAI expands to `srai rd′, rd′, shamt[5:0]`.

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



C.ANDI is a CB-format instruction that computes the bitwise AND of of the value in register *rd′* and the sign-extended 6-bit immediate, then writes the result to *rd′*. C.ANDI expands to `andi rd′, rd′, imm[5:0]`.

Integer Register-Register Operations

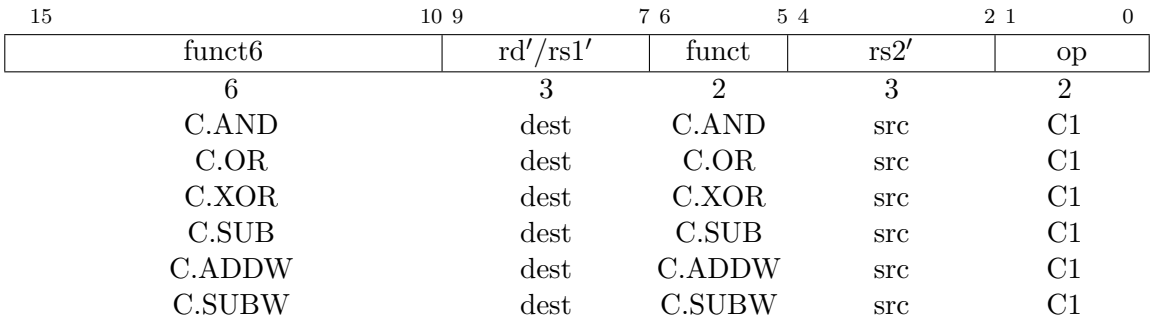


These instructions use the CR format.

C.MV copies the value in register *rs2* into register *rd*. C.MV expands into `add rd, x0, rs2`.

C.MV 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`.



These instructions use the CS format.

C.AND computes the bitwise AND of the values in registers *rd′* and *rs2′*, then writes the result to register *rd′*. C.AND expands into `and rd′, rd′, rs2′`.

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

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

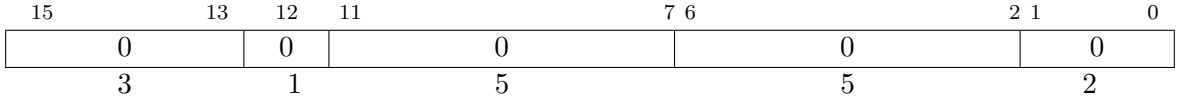
C.SUB subtracts the value in register *rs2′* from the value in register *rd′*, then writes the result to register *rd′*. C.SUB expands into `sub rd′, rd′, rs2′`.

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

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

This group of six instructions do not provide large savings individually, but do not occupy much encoding space and are straightforward to implement, and as a group provide a worthwhile improvement in static and dynamic compression.

Defined Illegal Instruction

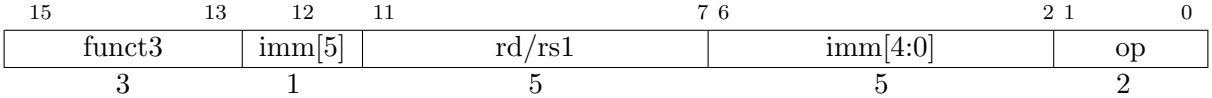


15–13	12	11–7	6–2	1–0
0	0	0	0	0
3	1	5	5	2
0	0	0	0	0

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

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

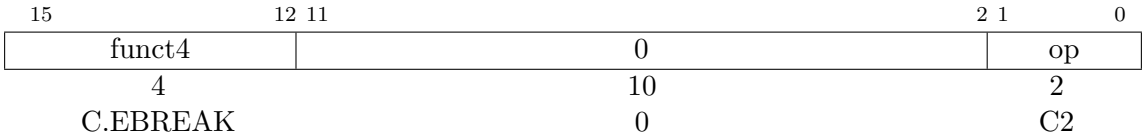
NOP Instruction



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

C.NOP is a CI-format instruction that does not change any user-visible state, except for advancing the `pc`. C.NOP is encoded as `c.addi x0, 0` and so expands to `addi x0, x0, 0`.

Breakpoint Instruction



Debuggers can use the C.EBREAK instruction, which expands to `ebreak`, to cause control to be transferred back to the debugging environment. C.EBREAK shares the opcode with the C.ADD instruction, but with *rd* and *rs2* both zero, thus can also use the CR format.

12.6 Usage of C Instructions in LR/SC Sequences

On implementations that support the C extension, compressed forms of the I instructions permitted inside LR/SC sequences can be used while retaining the guarantee of eventual success, as described in Section 7.2.

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.

12.7 RVC Instruction Set Listings

Table 12.3 shows a map of the major opcodes for RVC. Opcodes with the lower two bits set correspond to instructions wider than 16 bits, including those in the base ISAs. Several instructions are only valid for certain operands; when invalid, they are marked either *RES* to indicate that the opcode is reserved for future standard extensions; *NSE* to indicate that the opcode is reserved for non-standard extensions; or *HINT* to indicate that the opcode is reserved for future standard microarchitectural hints. Instructions marked *HINT* must execute as no-ops on implementations for which the hint has no effect.

The HINT instructions are designed to support future addition of microarchitectural hints that might affect performance but cannot affect architectural state. The HINT encodings have been chosen so that simple implementations can ignore the HINT encoding and execute the HINT as a regular operation that does not change architectural state. For example, C.ADD is a HINT if the destination register is `x0`*, where the five-bit rs2 field encodes details of the HINT. However, a simple implementation can simply execute the HINT as an add to register* `x0`*, which will have no effect.*

inst[15:13] / inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4SPN	FLD FLD LQ	LW	FLW LD LD	*Reserved*	FSD FSD SQ	SW	FSW SD SD	RV32 RV64 RV128
01	ADDI	JAL ADDIW ADDIW	LI	LUI/ADDI16SP	MISC-ALU	J	BEQZ	BNEZ	RV32 RV64 RV128
10	SLLI	FLDSP FLDSP LQ	LWSP	FLWSP LDSP LDSP	J[AL]R/MV/ADD	FSDSP FSDSP SQ	SWSP	FSWSP SDSP SDSP	RV32 RV64 RV128
11	>16b								

Table 12.3: RVC opcode map

Tables 12.4–12.6 list the RVC instructions.

15 14 13	12 11 10	9 8 7	6 5	4 3 2	1 0	
000	0			0	00	*Illegal instruction*
000	nzuimm[5:4|9:6|2|3]			rd′	00	C.ADDI4SPN *(RES, nzuimm=0)*
001	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.FLD *(RV32/64)*
001	uimm[5:4|8]	rs1′	uimm[7:6]	rd′	00	C.LQ *(RV128)*
010	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.LW
011	uimm[5:3]	rs1′	uimm[2|6]	rd′	00	C.FLW *(RV32)*
011	uimm[5:3]	rs1′	uimm[7:6]	rd′	00	C.LD *(RV64/128)*
100	—				00	*Reserved*
101	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.FSD *(RV32/64)*
101	uimm[5:4|8]	rs1′	uimm[7:6]	rs2′	00	C.SQ *(RV128)*
110	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.SW
111	uimm[5:3]	rs1′	uimm[2|6]	rs2′	00	C.FSW *(RV32)*
111	uimm[5:3]	rs1′	uimm[7:6]	rs2′	00	C.SD *(RV64/128)*

Table 12.4: Instruction listing for RVC, Quadrant 0.

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

Table 12.5: Instruction listing for RVC, Quadrant 1.

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

Table 12.6: Instruction listing for RVC, Quadrant 2.

Chapter 13

"B" Standard Extension for Bit Manipulation, Version 0.0

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

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

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

Chapter 14

"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 15

"T" Standard Extension for Transactional Memory, Version 0.0

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

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

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

Chapter 16

"P" Standard Extension for Packed-SIMD Instructions, Version 0.1

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.

In this chapter, we outline a standard packed-SIMD extension for RISC-V. We've reserved the instruction subset name "P" for a future standard set of packed-SIMD extensions. Many other extensions can build upon a packed-SIMD extension, taking advantage of the wide data registers and datapaths separate from the integer unit.

Packed-SIMD extensions, first introduced with the Lincoln Labs TX-2 [9], have become a popular way to provide higher throughput on data-parallel codes. Earlier commercial microprocessor implementations include the Intel i860, HP PA-RISC MAX [19], SPARC VIS [29], MIPS MDMX [12], PowerPC AltiVec [8], Intel x86 MMX/SSE [24, 26], while recent designs include Intel x86 AVX [20] and ARM Neon [11]. We describe a standard framework for adding packed-SIMD in this chapter, but are not actively working on such a design. In our opinion, packed-SIMD designs represent a reasonable design point when reusing existing wide datapath resources, but if significant additional resources are to be devoted to data-parallel execution then designs based on traditional vector architectures are a better choice and should use the V extension.

A RISC-V packed-SIMD extension reuses the floating-point registers (`f0-f31`). These registers can be defined to have widths of FLEN=32 to FLEN=1024. The standard floating-point instruction subsets require registers of width 32 bits ("F"), 64 bits ("D"), or 128 bits ("Q").

It is natural to use the floating-point registers for packed-SIMD values rather than the integer registers (PA-RISC and Alpha packed-SIMD extensions) as this frees the integer registers for control and address values, simplifies reuse of scalar floating-point units for SIMD floating-point execution, and leads naturally to a decoupled integer/floating-point hardware design. The floating-point load and store instruction encodings also have space to handle wider packed-SIMD registers. However, reusing the floating-point registers for packed-SIMD values does make it more difficult to use a recoded internal format for floating-point values.

The existing floating-point load and store instructions are used to load and store various-sized words from memory to the `f` registers. The base ISA supports 32-bit and 64-bit loads and stores, but the LOAD-FP and STORE-FP instruction encodings allows 8 different widths to be encoded as shown in Table 16.1. When used with packed-SIMD operations, it is desirable to support non-naturally aligned loads and stores in hardware.

width field	Code	Size in bits
000	B	8
001	H	16
010	W	32
011	D	64
100	Q	128
101	Q2	256
110	Q4	512
111	Q8	1024

Table 16.1: LOAD-FP and STORE-FP width encoding.

Packed-SIMD computational instructions operate on packed values in `f` registers. Each value can be 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit, and both integer and floating-point representations can be supported. For example, a 64-bit packed-SIMD extension can treat each register as 1×64-bit, 2×32-bit, 4×16-bit, or 8×8-bit packed values.

Simple packed-SIMD extensions might fit in unused 32-bit instruction opcodes, but more extensive packed-SIMD extensions will likely require a dedicated 30-bit instruction space.

Chapter 17

"V" Standard Extension for Vector Operations, Version 0.3-DRAFT

This chapter presents a proposal for the RISC-V vector instruction set extension. The vector extension supports a configurable vector unit, to tradeoff the number of architectural vector registers and supported element widths against available maximum vector length. The vector extension is designed to allow the same binary code to work efficiently across a variety of hardware implementations varying in physical vector storage capacity and datapath spatial and/or temporal parallelism. 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.

The vector extension is based on the style of vector register architecture introduced by Seymour Cray in the 1970s, as opposed to the earlier packed SIMD approach, introduced with the Lincoln Labs TX-2 in 1957 and now adopted by most other commercial instruction sets.

The vector instruction set contains many features developed in earlier research projects, including the Berkeley T0 [] and VIRAM [] vector microprocessors, the MIT Scale vector-thread processor [], and the Berkeley Maven [] and Hwacha [] projects.

17.1 Vector Unit State

The additional vector unit architectural state consists of 32 vector data registers (`v0`–`v31`), 8 vector predicate registers (`vp0`-`vp7`), and an XLEN-bit WARL vector length CSR, `vl`. In addition, the current configuration of the vector unit is held in a set of vector configuration CSRs (`vdcfg0`–`vdcfg7` and `vnp`), as described below. The implementation determines an available *maximum vector length* (MVL) for the current configuration held in the `vdcfg` and `vnp` registers. There is also a 3-bit fixed-point rounding mode CSR `vxrm`, and a single-bit fixed-point saturation status CSR `vxsat`.

Future vector extensions using wider instruction encodings can support more architectural vector registers. For example, 256 architectural vector registers in a 64 bit encoding.

The `vcs` CSR alias provides combined access to the `vl`, `vxrm`, `vxsat`, and `vnp` fields to reduce context switch time. The `vcs` register also includes a configuration mode field to support future extended configuration modes.

CSR name	Number	Base ISA	Description
vcs	TBD	RV32, RV64, RV128	Vector control-status register
vl	TBD	RV32, RV64, RV128	Active vector length
vxrm	TBD	RV32, RV64, RV128	Vector fixed-point rounding mode
vxsat	TBD	RV32, RV64, RV128	Vector fixed-point saturation flag
vnp	TBD	RV32, RV64, RV128	Number of vector predicate registers
vdcfg0	TBD	RV32, RV64, RV128	Vector data register configuration
vdcfg1	TBD	RV32	
vdcfg2	TBD	RV32, RV64	
vdcfg3	TBD	RV32	
vdcfg4	TBD	RV32, RV64, RV128	
vdcfg5	TBD	RV32	
vdcfg6	TBD	RV32, RV64	
vdcfg7	TBD	RV32	

Table 17.1: Vector extension CSRs.

Discussion: *The components of vcs might not need separate CSR addresses, depending on how they're accessed via other non-CSR instructions.*

The vector unit must be configured before use. Each architectural vector data register (v0–v31) is configured with the bit width and type of each element of that vector data register, or can be disabled to free physical vector storage for other architectural vector data registers. The number of available vector predicate registers can also be set independently, from 0 to 8.

Several earlier vector machines had the ability to configure physical vector register storage into a larger number of short vectors or a shorter number of long vectors, in particular the Fujitsu VP series [21].

The available MVL depends on the configuration setting, but MVL must always have the same value for the same configuration parameters on a given implementation. Implementations must provide an MVL of at least four elements for all supported configuration settings.

Specifying a minimum MVL allows operations on known-short vectors to be expressed without requiring stripmining instructions.

Discussion: *Both min(MVL) and max(MVL) might be better expressed as part of a profile.*

Each vector data register's current configuration is described with an 8-bit encoding split into a 3-bit current maximum-width field vemaxw$n$ and a 5-bit type field vetype$n$, held in the vdcfg$x$ CSRs. The configuration state is also accessible via other specialized vector configuration instructions.

17.2 Element Datatypes and Width

The datatypes and operations supported by the V extension depend upon the base scalar ISA and supported extensions, and may include 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit integer and

Supported Fixed-Point Types	
RV32I	X8, U8, X16, U16, X32, U32
RV64I	X8, U8, X16, U16, X32, U32, X64, U64
RV128I	X8, U8, X16, U16, X32, U32, X64, U64, X128, U128

Supported Floating-Point Types	
F	F16, F32
FD	F16, F32, F64
FDQ	F16, F32, F64, F128

Table 17.2: Supported data element types depending on base integer ISA and supported floating-point extensions. Signed and unsigned integers are given separate types (e.g, X32 is signed 32-bit value, whereas U32 is an unsigned integer value). Note that supporting a given floating-point width mandates support for all narrower floating-point widths.

fixed-point data types (X8/U8, X16/U16, X32/U32, X64/U64, and X128/U128 respectively, where U indicates unsigned), and 16-bit, 32-bit, 64-bit, and 128-bit floating-point types (F16, F32, F64, and F128 respectively). When the V extension is added, it must support the vector data element types implied by the supported scalar types as defined by Table 17.2. The largest element width supported:

$$ELEN = max(XLEN, FLEN)$$

Compiler support for vectorization is greatly simplified when any hardware-supported data types are supported by both scalar and vector instructions.

Future vector extensions might expand the set of supported datatypes, including custom application-specific datatypes.

Adding the vector extension to any machine with floating-point support adds support for the IEEE standard half-precision 16-bit floating-point data type. This includes a set of scalar half-precision instructions described in Section **??**. The scalar half-precision instructions follow the template for other floating-point precisions, but using the hitherto unused *fmt* field encoding of `10`.

There is interest in splitting off the scalar half-precision instructions into their own named extension.

17.3 Vector Element Width (`vemaxw`n)

The current maximum element width for vector data register n is held in a three-bit field, `vemaxw`n, encoded as shown in Table 17.3.

Future extensions might increase the supported vector element widths beyond those of the base scalar ISA, or support smaller non-power-of-2 widths. At least one of the remaining width values should be reserved to support a width-encoding escape to support this larger range of width values.

Width	Encoding
Disabled	000
8	100
16	101
32	110
64	111
128	011

Table 17.3: Encoding of vector element maximum-width fields `vemaxw0`–`vemaxw31`. All other values are reserved.

Three broad classes of implementation can be distinguished by how they handle `vemaxwn` *settings.*

The simplest is max-width-per-implementation *(MWPI), where the vector unit is organized in fixed ELEN-width physical lanes, and changes to* `vemaxwn` *settings simply cause portions of the physical registers and datapath to be disabled for operations narrower than ELEN bits.*

The next most complex implementation, max-width-per-configuration *(MWPC), uses the maximum width across all* `vemaxwn` *settings in a dynamic configuration to divide the physical register storage and datapaths. For example, a MWPC machine with ELEN=64 might subdivide physical lanes into 32-bit datapaths if no* `vemaxwn` *setting is greater than 32. Operations on sub-32-bit quantities would disable appropriate portions of the physical registers and functional units in each 32-bit lane. Several early vector supercomputers, including the CDC Star-100 [?], provided a similar facility to divide 64-bit physical vector lanes into narrower 32-bit lanes.*

The most complex implementations are max-width-per-register *(MWPR), which reduce wasted space in the physical register files by packing elements in each vector register according to the individual* `vemaxwn` *settings and which within one configuration can execute instructions with narrower datatypes at higher rates than for wider datatypes. The Berkeley Hwacha vector engine [?, ?] is an example microarchitecture with this property.*

Any write to any `vemaxw`*n* field configures the entire vector unit and causes all vector data registers to be zeroed and all vector predicate registers to be set, and the vector length register `vl` to be set to the maximum supported vector length.

Vector registers are zeroed on reconfiguration to prevent security holes and to avoid exposing differences between how different implementations manage physical vector register storage.

In-order implementations will probaby use a flag bit per register to mux in 0 instead of garbage values on each source until it is overwritten. For in-order machines, vector lengths less than MVL complicate this zeroing, but these cases can be handled by adding a zero bit per element or element group. Machines with vector register renaming can just initialize the rename table to point entries at a physical zero register.

If a vector data register is disabled, then any vector instruction that attempts to access that vector data register will raise an illegal instruction exception. Attempting to write any `vemaxw`*n* with an unsupported value will raise an illegal instruction exception.

17.4 Vector Element Type (`vetype`*n*)

The current element type of vector data register n is held in a five-bit `vetype`*n* field encoded as shown in Table 17.4. The element type `vetype`*n* of a vector data register is constrained to have

equal or lesser width than the value in the corresponding `vemaxw`*n* field. A write to a `vetype`*n* field zeros the associated vector data register `v`*n*, but leaves other vector unit state undisturbed. Changes to `vetype`*n* do not alter MVL.

Type	`vemaxw` equivalent	`vetype` encoding
Disabled	000	00000
Floating-Point types		
F16	101	01101
F32	110	01110
F64	111	01111
F128	011	01011
Signed integer and fixed-point types		
X8	100	10100
X16	101	10101
X32	110	10110
X64	111	10111
X128	011	10011
Unsigned integer and fixed-point types		
U8	100	11100
U16	101	11101
U32	110	11110
U64	111	11111
U128	011	11011

Table 17.4: Encoding of `vetype` fields. All other values are reserved. The middle column shows the value that will be written to `vemaxw`*n* for configuration instructions that write both `vetype`*n* and `vemaxw`*n* fields. For these standard types, `vemaxw`*n* follows the low three bits of `vetype`*n*. The value of `vetype`*n* can be changed independently of `vemaxw`*n* provided the required element width is less than or equal to `vemaxw`*n*.

Vector data registers have both a maximum element width and a current element data type to allow the same vector data register to be changed to different types during execution provided the maximum width is not exceeded. This reduces register pressure and helps support vector function calls, where the caller does not know the types needed by the callee, as described below.

The set of supported types might be greatly increased with future extensions. For example (and not limited to), new scalar types in new number systems, a complex type with real and imaginary components, a key-value type, or an application-specific structure type with multiple consitituent fields. Auxiliary type configuration state might be required in these cases.

Attempting to write an unsupported type or a type that requires more than the current `vemaxw` width to a `vetype` field will raise an illegal instruction exception.

Implementations must still raise an exception for a `vetype`*n setting that is greater than the architectural* `vemaxw`*n width, even if they internally implement a larger physical* `vemaxw`*n that could accomodate the* `vetype`*n request.*

Discussion: *We can either have 1) implementations raise exceptions whenever illegal values*

are written to `vemaxw` *and* `vetype` *fields (current design), 2) raise exceptions at use if config holds illegal values, 3) make the fields WARL so silently reduce to supported types with no exceptions. Option 2 could complicate vector unit context switch code by having more cases to check, while Option 3 could make debugging more difficult by allowing code to run with reduced precision or incorrect types.*

17.5 Vector Predicate Configuration Register (`vnp`)

The `vnp` CSR holds a single 4-bit value giving the number of enabled architectural predicate registers, between 0 and 8. Any write to `vnp` zeros all vector data registers, sets all bits in visible vector predicate registers, and sets the vector length register `vl` to the maximum supported vector length. Attempting to write a value larger than 8 to `vnp` raises an illegal instruction exception.

Discussion: *The number of vector predicate registers supported in base ISA could be changed. The base encoding could support up to 32 predicate registers, but it is not clear these would be used frequently enough to warrant increased the architectural cost for all implementations.*

roger: we should force the minimum number of vp registers to 2, so that vp0 and vp1 are always available to the compiler. This would work nicer with the encoding that has a bit that allows selecting vp0 or vp1.

When `vnp` is 0, any instruction that reads a vector predicate register other than `vp0` will raise an illegal instruction exception, while reads of `vp0` will return all ones to provide unpredicated execution. When `vnp` is 0, any instruction that attempts to write any vector predicate register will raise an illegal instruction exception.

17.6 Vector Data Configuration Registers (`vdcfg0`–`vdcfg7`)

The vector data register configuration requires 256 bits of state (32 vector data registers each with a 3-bit `vemaxw`n field and a 5-bit `vetype`n field), and is held in the `vdcfg CSRs`.

RV128 has two vector configuration CSRs: `vdcfg0` holds configuration data for `v0`–`v15` with bits $8n$ to $8n+4$ holding `vetype`n and bits $8n+5$ to $8n+7$ holding `vemaxw`n, while `vdcfg4` similarly holds configuration data for `v16`–`v31`.

In RV64, the `vdcfg2` CSR provides access to the upper 64 bits of `vdcfg0` and `vdcfg6` provides access to the upper 64 bits of `vdcfg4`. In RV32, the `vdcfg1`, `vdcfg3`, `vdcfg5` and `vdcfg7` CSRs provides access to the upper bits of `vdcfg0`, `vdcfg2`, `vdcfg4` and `vdcfg6` respectively.

Any CSR write to a `vdcfg`x register zeros all `vdcfg`y registers, for $y > x$, and also zeros the `vnp` register. As a result configuration data should be written from the `vdcfg0` CSR upwards, followed by the `vnp` setting if non-zero.

Zeroing higher-numbered `vdcfg`y *registers allows more rapid reconfiguration of the vector register file via CSR writes, and provides backward-compatibility for extensions that increase the number of possible architectural vector registers. This choice does prevent the use of CSRRW instructions to swap the configuration context.*

Additional instructions are provided to support more rapid changes to the vector unit configuration as described below. These directly affect the `vemaxw`*n and* `vetype`*n fields and do not necessarily have the same side effects as the CSR writes through the* `vdcfg`*n addresses.*

17.7 Legal Vector Unit Configurations

To simplify hardware configuration calculations and to reduce software context-switch complexity, vector unit configurations are constrained to have non-disabled architectural vector registers numbered contiguously starting at `v0`. Also, `vemaxw`m must be greater than or equal to `vemaxw`n, for $m > n$, i.e., configured element widths must increase monotonically with architectural vector register number. An exception will be raised if any instruction tries to change `vemax`n in a way that violates this constraint.

During a software vector-context save, the software handler can stop searching for active architectural registers after encountering the first disabled vector register. Hardware to calculate physical register allocation might be slightly simplified with this constraint, and might be able to pack register storage more tightly with monotonically increasing element size.

In a vector-function calling convention, higher-numbered registers are usually made available to the callee, and must usually be a wider, often ELEN-width, element. The context that configures the vector unit might have known-narrower element types and can save storage by confguring the lower-numbered architectural vector registers accordingly.

17.8 Vector Instruction Formats

The instruction encoding is a work in progress.

An important design goal was that the base vector extension fit within a few major opcodes of the 32-bit encoding. It is envisioned that future vector extensions will use 48-bit or 64-bit encodings to increase both the opcode space and the set of architectural registers. The 64-bit vector encoding would support 256 architectural vector registers and orthogonal specification of a predicate register in each instruction.

Vector arithmetic and vector memory instructions are encoded in new variants of the R-format, shown in Figure 17.1. Both new formats use one bit to hold a *vp* field, which usually controls the predicate register in use, either `vp0` or `vp1`. The VR4 form is used for fused multiply-add instructions. The existing RISC-V instruction formats are used for other vector-related instructions, such as the vector configuration instructions.

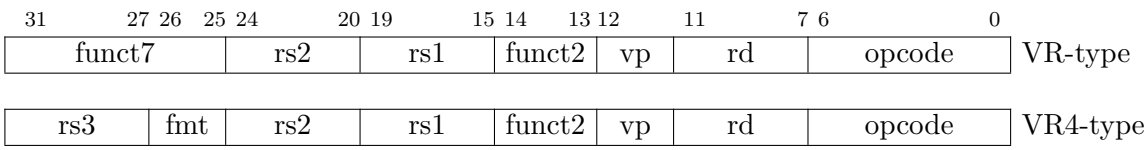


Figure 17.1: New V extension instruction formats.

Most vector instructions are available in both vector-vector and vector-scalar variants. Vector-vector instructions take the first operand from the vector register specified by *rs1* and the second operand from the vector register specified by *rs2*.

For vector-scalar operations, the *rs1* field specifies the scalar register to be accessed. For most vector-scalar instructions, the type of the vector operand specified by *rs2* indicates whether the integer or floating-point scalar register file is accessed using the *rs1* register specifier.

Some non-commutative vector-scalar instructions (such as sub) are provided in two forms, with the scalar value used as the second operand.

The rs1 *field is used to provide the scalar operand because in the base encoding, whenever an instruction has a single scalar source operand, it is encoded in the* `rs1` *field.*

17.9 Polymorphic Vector Instructions

The vector extension uses a polymorphic instruction encoding where the opcode is combined with the types of the source and destination registers to determine the operation to be performed. For example, an ADD opcode will perform a 32-bit integer vector-vector add if both vector source operands and the vector destination register are 32-bit integers, but will perform a 16-bit floating-point vector-vector operation if both vector source operands and the vector destination are 16-bit floats.

The polymorphic encoding also naturally supports operations with mixed precisions on the input and output, and also supports extending the instruction set with new types without necessarily increasing the opcode space.

Not all combinations of source and destination argument types need be supported. The base vector extension mandates only that implementations provide a subset of combinations of types on inputs and outputs. Table 17.5 shows the general rules for integer and floating-point instructions, but the detailed instruction listing should be consulted for accurate information.

A general rule in the base vector instruction set is that the destination precision is never less than any source operand, except for explicit type-conversion instructions. Another general rule is that the input operands can only be the same width or half the width of the destination operand except for the scalar operand in integer vector-scalar instructions, which is always XLEN wide. Also, src2 is never larger than src1 or src3.

Integer computations of mixed-precision values always aligns values by their LSB, and sign or zero-extends any smaller value according to its type. The result is truncated to fit in the destination type. Note a scalar integer value is already XLEN bits wide, and as wide as any possible integer vector value.

Floating-point computations on mixed-precision values acts as if the calculations are performed exactly then rounded once to the destination format.

17.10 Rapid Configuration Instructions

It can take several CSR instructions to set up the `vdcfg` and `vnp` CSRs for a given configuration. Specialized configuration instructions are provided to quickly set up common configurations in the

Src1	Src2	Src3	Dest	Example
Integer vector-scalar				
XLEN	X	-	X	64b + 32b → 32b
XLEN	X	-	2X	64b + 8b → 16b
Integer vector-vector				
X	X	-	X	32b + 32b → 32b
X	X	-	2X	16b + 16b → 32b
2X	X	-	2X	64b + 32b → 64b
Floating-point vector-scalar				
F	F	-	F	64b + 64b → 64b
F	F	F	F	32b × 32b + 32b → 32b
F	F	-	2F	32b + 32b → 64b
F	F	2F	2F	32b × 32b + 64b → 64b
Floating-point vector-vector				
F	F	-	F	32b + 32b → 32b
F	F	-	2F	16b + 16b → 32b
2F	F	-	2F	64b + 32b → 64b
F	F	F	F	64b × 64b + 64b → 64b
F	F	2F	2F	16b × 16b + 32b → 32b

Table 17.5: General rules for supported types per instruction in base vector extension. X represents the number of bits in an integer type and F represents the number of bits in a floating-point type. Individual instruction types will provide more detailed listings. Note that the type of a scalar floating-point operand can never be different from that of the vector in Src2, hence the Src1=2F case is missing from vector-scalar operations.

`vdcfg` and `vnp` CSRs.

The `vsetdcfg` instruction takes a scalar register value encoded as shown in Figure 17.2, and returns the corresponding MVL in the destination register. The `vsetdcfg` and `vsetdcfgi` instructions also clear the `vnp` register, so no predicate registers are allocated.

Discussion: *For now, only a 32-bit value supporting up to three different vector data types is supported by the* `vsetdcfg` *instruction. RV64 and RV128 could support larger number of types, though it's not clear if the hardware cost (area, latency) to support a larger number of different types is justified.*

The `vsetdcfg` value specifies how many vector registers of each datatype are allocated, and is divided into a 2-bit mode field and pairs of 5-bit fields for each data type in the configuration.

The 2-bit mode field indicates the configuration mode of the vector unit and is zero for the base vector extension.

The standard vector extension operating mode configures the vector unit into some number of vector registers, each with some number of elements of types supported by the scalar unit.

At least one alternative mode is planned, where the vector unit is configured as some number of registers each holding a single large element, e.g., 256 bits. This would be the base for cryptographic operations, or other coprocessors that operated on large structures.

Other modes can be used to reconfigure the vector unit register file and functional units for other domain-specific purposes.

Each datatype pair contains a 5-bit `type`x value encoded as a `vetype`n value, and a 5-bit `ntype`x for the number of registers to allocate for that type. If the `type0` field is non-zero, the `vsetdcfg` instruction will configure the first `ntype0` vector data registers to have `vetype`n values of `type0` with `vemaxw`n values set accordingly as shown in Table 17.4. If the `type0` value is 0, the datatype pair is skipped. If the `type1` field is non-zero, then the next `ntype1` vector registers are configured to be of the type given in `type1`. Similarly for the `type2` pair.

A value of zero in a `type`x field indicates this datatype pair should be ignored. A value of zero in a `ntype`x field indicates 32 registers should be allocated for the corresponding type.

Zero values are skipped to simplify setting a configuration with two different data types, where a single LUI instruction can set the upper 20 bits leaving the low bits zero.

A single 12-bit immediate value is sufficient to create a configuration with some number of vector registers with a single given datatype.

A compressed C.LI with a zero-extended 5-bit immediate can create a configuration with 32 vector registers of a given datatype.

				mode		
`type2`	`ntype2`	`type1`	`ntype1`	0	`type0`	`ntype0`
5	5	5	5	2	5	5

Figure 17.2: Format of the `vsetdcfg` value. The value contains three pairs of a 5-bit type and a 5-bit number of registers to create of that type. A value of 0 for the number of a type indicates that 32 registers should be allocated. A value of 0 for the type indicates this pair should be skipped. The types must be of monotonically increasing size from type0 to type2.

A corresponding `vsetdcfgi` instruction takes a 12-bit immediate value to set the configuration instead of a scalar value, but otherwise is identical to the `vsetcfgd` instruction.

Discussion: *It is not clear how many immediate bits will be made available for the* `vsetdcfgi` *instruction. If encoding space is available for both 12 immediate bits and a source register specifier, then* `vsetdcgfi` *can be defined to read the source register, OR in the bits in the immediate, then create a configuration. In this case, there is no need for a separate* `vsetdcfg` *instruction.*

The configuration value given must result in a legal configuration or else an illegal instruction exception will be raised.

If a zero argument is given to `vsetdcfg` the vector unit will be disabled and the value 0 will be returned for MVL. This instruction (`vsetdcfg x0, x0`) is given the assembly pseudo-code `vdisable`.

Separate `vsetpcfg` and `vsetpcfgi` instructions are provided that write the source value to the `vnp` register and return the new MVL. These writes also clear the vector data registers, set all bits in the allocated predicate registers, and set `vl`=MVL. A `vsetpcfg` or `vsetpcfgi` instruction can be used after a `vsetdcfg` to complete a reconfiguration of the vector unit.

Discussion: *If* `vnp` *is made accessible as a separate CSR, the* `setpcfg` *and* `setpcfgi` *instructions are less useful. The only advantage over a CSR instruction is that they return MVL, which is rarely needed, and which can be obtained via that* `setvl` *instruction.*

17.11 Vector-Type-Change Instructions

To quickly change the individual types of a vector register, `vetyperw` and `vetyperwi` instructions are provided to change the type of the specified vector data register to the given scalar register value or 5-bit immediate value respectively, while returning the previous type in the destination scalar register.

A vector convert instruction, described below, can simultaneously convert a source vector register into a new type, and set that type in the destination vector register.

17.12 Vector Length

The active vector length is held in the XLEN-bit WARL vector length CSR `vl`, which can only hold values between 0 and MVL inclusive. Any writes to the configuration registers (`vdcfg`x or `vnp`) cause `vl` to be initialized with MVL. Changes to `vetype`n via vector-type-change instructions do not affect `vl`.

The active vector length is usually set via the `setvl` instruction. The source argument to the `setvl` is the requested application vector length (AVL) as an unsigned XLEN-bit integer. The `setvl` instruction calculates the value to assign to `vl` according to Table 17.6. The result of this calculation is also returned as the result of the `setvl` instruction.

Earlier drafts encoded `setvl` *using a modified CSRRW instruction whereas it is now encoded as a separate new instruction.*

AVL Value	`vl` setting
AVL $\geq$ 2 MVL	MVL
2 MVL $>$ AVL $>$ MVL	$\lceil$AVL/2$\rceil$
MVL $\geq$ AVL	AVL

Table 17.6: Operation of `setvl` instruction to set vector length register `vl` based on requested application vector length (AVL) and current maximum vector length (MVL).

The rules for setting the `vl` register help keep vector pipelines full over the last two iterations of a stripmined loop. This version of the rules guarantees monotonically decreasing vector lengths. Similar rules were previously used in Cray-designed machines [7].

Discussion: *There are multiple possible rules for setting VL, and we could give implementations freedom to use different VL setting rules.*

The idea of having implementation-defined vector length dates back to at least the IBM 3090 Vector Facility [5], which used a special "Load Vector Count and Update" (VLVCU) instruction to control stripmine loops. The `setvl` instruction included here is based on the simpler `setvlr` instruction introduced by Asanović [4].

The `setvl` instruction is typically used at the start of every iteration of a stripmined loop to set the number of vector elements to operate on in the following loop iteration. The current MVL can be obtained from a vector configuration instruction, or by performing a `setvl` with a source argument that has all bits set (largest unsigned integer).

When `vl` is less than MVL, vector instructions will set all elements in the range [`vl`:MAXVL-1] in the destination vector data register or destination vector predicate register to zero.

Requring zeroing of elements past the current active vector length simplifies the design of units with renamed vector data registers. If the specification left destination elements unchanged, renaming implementations would have to copy the tail of the old destination register to the newly allocated destination register. Alternatively, specifying the tail to be undefined will expose implementation differences and possibly cause a security hole.

Implementations that do not support renaming, will have to zero the tail of a vector, but this can reuse the mechanism that is already required to initialize all vector data registers to zero on reconfiguration, for example, by having a zero bit on each element or element group.

No element operations are performed for any vector instruction when `vl`=0.

Two possible choices are to 1) require destination registers to be completely zeroed when `vl`=0, or 2) no changes to the destination registers. Option 2 is currently chosen as this will prevents unnecessary work in some implementations, and option 1 does not provide a clear advantage beyond seeming more consistent with `vl`¿0 case.

17.13 Predicated Execution

All vector instructions in the base vector instruction set have a single bit to select either `vp0` or `vp1` as the active predicate register.

```
        # Vector-vector 32-bit add loop.
        # a0 holds N
        # a1 holds pointer to result vector
        # a2 holds pointer to first source vector
        # a3 holds pointer to second source vector
        li t0, (2<<VNTYPE0|VREGF32)
        vsetdcfg t0     # Configure with two 32-bit float vectors

loop:   setvl t0, a0    # Set length, get how many elements in strip
        vld v0, a2      # Load first vector
        sll t1, t0, 2   # Multiply length by 4 to get bytes
        add a2, t1      # Bump pointer
        vld v1, a3      # Load second vector
        add a3, t1      # Bump pointer
        vadd v0, v1     # Add elements
        sub a0, t0      # Decrement elements completed
        vst  v0, a1     # Store result vector
        add a1, t1      # Bump pointer
        bnez a0, loop   # Any more?

        vdisable        # Turn off vector unit
```

Figure 17.3: Example vector-vector add loop.

The 32-bit base encoding does not leave room for a fully orthogonal predicate register specifier. A single bit is dedicated to the predicate register specification, and is used to select between two active predicate registers, `vp0` *or* `vp1`*. An alternative scheme would have used the bit to select between* `vp0` *and unpredicated (all elements active). However, given the ease of setting all predicate bits in a vector predicate register with a single predicate instruction, the current scheme provides more flexibility.*

When there are no vector predicate registers enabled, `vp0` *returns all set bits when read. So, the assembler convention is to assume* `vp0` *as the predicate register when no predicate register is explicitly given. The assembler can support a strict operands option to require the vector predicate register is explicitly specified.*

At element positions where the selected predicate register bit is zero, the corresponding vector element operation has no effect (does not change architectural state or generate exceptions), except to write a zero to the element position in the destination vector register.

Discussion: *The previous proposal (undisturb) left the destination vector unchanged at element positions where the predicate bit is false, whereas the current plan-of-record (zero) writes zero to the destination where the predicate bit is false.*

The advantage of the undisturb option is that it can require fewer instructions and fewer architectural registers for many common code sequences. For in-order machines without register renaming, the undisturb operation simply disables writes to the destination elements, except for vector registers that have not been written since configuration time. Typically an extra zero bit per vector register or element group will be added to represent a zeroed register instead of actually zeroing state at configuration time. For predicated undisturb writes to these uninitialized registers, the predicated false elements must be explicity written with zeros on each element group

and the zero bit is then cleared down. However, in a machine with vector register renaming, undisturb does imply an additional read of the original destination register to write the value into the new physical destination register when the predicate is false. This additional read port will often be cheaper than in a scalar machine as vector machines often time-multiplex read ports, and the additional read can be skipped when the predicate registers are disabled (`vnp`*=0) or when the source is known to be zero after configuration, but still adds complexity to a design.*

The advantage of the zero option is that a machine with vector register renaming does not need to read the original destination vector register and so a read port is saved. The disadvantage of the zero option is that more instructions and architectural registers are required for common code sequences, and simpler microarchitectures without register renaming are penalized by requiring longer code sequences and greater register pressure. In particular, vector merge instructions are required to collect results from two divergent control paths, and each vector merge has to read two vector values and write a vector result. Whether the zero option saves total register file traffic in an register-renamed microarchitecture depends on the ratio of a) internal temporary writes, to b) writes creating values that are live out of each basic block, and also to the frequency of control flow merges.

Overall, the zero option removes significant complexity from the renamed machines while reducing efficiency somewhat for the non-renamed machines, and is the current plan-of-record.

The following sections are preliminary notes.

17.14 Predicate Operations

All the standard logical operations are defined on predicate registers.

The predicate operations have effectively three inputs, the two register specifiers rs1 and rs2, plus the predicate specifier `vp0` *or* `vp1`*. There are 256 possible logic operations on three bits of input, so can specify with an 8-bit immediate providing the lookup table.*

A predicate swap operation is defined to exchange the values of two predicate registers. This is used to work around the lack of predicate register specifiers in the base vector ISA.

```
vpswap vp0, vp5  # Exchange values in vp0 and vp5.
```

The predicate swap can be performed with just rename table updates in a renamed architecture. Non-renamed machines will have to explicitly copy the values.

Discussion: *Not clear if the swap is really needed, or if explicit moves into* `vp0` *and* `vp1` *will suffice.*

The predicate operations include operations to support software vector-length speculation for vectorization of while loops.

The general scheme is described in Chapter 6 of [4].

17.15 Vector Load/Store Instructions

Three vector load/store addressing modes are supported, unit-stride, constant stride, and indexed (scatter/gather). Each addressing mode has a 7-bit unsigned immediate offset that is scaled by the element type.

The unit-stride address mode takes a scalar base byte address, adds the scaled immediate, then generates a contiguous set of element addresses for loads or stores.

The primary use of immediates in unit-stride loads is to generate overlapping unit-stride loads for convolution operations.

The constant-stride address mode takes a scalar base byte address, a stride value encoded in bytes, and adds a scaled immediate value.

The stride value is in bytes to allow a single stride register to be used to support operations on arrays-of-structures, where not all elements in each structure have the same size. The immediate value is still scaled by element size to increase reach, given that element types will be naturally aligned.

The indexed address mode takes a scalar base byte address and a vector of byte offsets. The scalar base address and the immediate value are added to element of the offset vector to give a vector of addresses used in a scatter/gather.

Indexed stores are provided in three types. Unordered, ordered, and reverse-ordered. The unordered indexed stores might update the same memory location from two different elements in an unspecified order. The ordered stores always update memory locations in increasing vector element order. The reverse-ordered stores always update memory locations in decreasing memory order.

The reverse-ordered stores support vectorization of software memory disambiguation techniques. A reverse-ordered store of element id into a hash table indexed by a hash on a store access address, followed by a read of the hash table using a load access address and a comparison against the original element id, will indicate if there's a potential RAW hazard with an earlier loop iteration.

Discussion: *Not clear if there is sufficient realizable improvement for supporting unordered stores over ordered stores.*

Vector loads/stores have a simple memory model, where each vector load/store is observed to complete sequentially in program order ony the local hart, i.e., a vector load on a hart will observe all earlier vector stores on the same hart, and no later vector stores.

Vector loads are available in a length-speculative form that writes predicate register `vp1` in addition to the destination vector data register. These instructions raise an illegal instruction exception if `vp1` is not configured. For elements that do not generate a permissions fault, the length-speculative vector loads operate as normally except to also clear the bit in `vp1`. If an element encounters a permission fault, a zero is written to the destination vector register element and the `vp1` bit is set to a 1. Implementations may treat elements past the first faulting element as also causing a fault even if they might not cause a permissions fault when accessed alone.

Once software determines the active vector length, it should check if any loads within the active vector length caused a fault, and in this case, generate a non-length-speculative load to trigger reporting of the error.

Length-speculative vector loads are required to vectorize while loops, with data-dependent exits (e.g. strlen).

The only faults ignored by the length-speculative vector loads are ones that would have resulted in a permissions violation. Page faults and other virtualization-related faults should be handled invisibly to the user thread by the execution environment.

A malicious program can use length-speculative vector loads to probe accessible address space without fear of a fatal fault.

17.16 Vector Select

A vector select produces a new result data vector by gathering elements from one source data vector at the element locations specified by a second source index vector. Data source and destination vector types must agree. The index vector can have any integer type. Legal element indices can range from 0 to current MAXVL. Indices out of this range raise an illegal instruction exception.

```
# vindices holds values from 0..MAXVL
vselect  vdest, vsrc, vindices
```

17.17 Reductions

Reductions are supported via a vector extract instruction that takes elements starting from the middle of one vector and places these at the beginning of a second vector register. This supports a recursive-halving reduction approach for any binary associative operator.

A similar vector register extract instruction was added to the Cray C90 after memory latency grew too large for the memory-memory reductions used in earlier Crays.

The vector unit microarchitecture can be optimized for the power-of-2 sized element offsets used for reductions.

17.18 Fixed-Point Support

Clip instruction supports scaling, rounding, and clipping to destination type. Rounding set by CSR fixed-point rounding mode (truncate, jam, round-up, round-nearest-even). Clipping set by CSR clip mode (wrap, saturate).

Add with average, rounding set by rounding mode.

Multiply with same size source and destination types, with some result scaling values (+1, 0, -1, -8?) and rounding and clipping according to CSR mode.

Accumulate with carry into predicate register to support larger precise dot-products.

17.19 Optional Transcendental Support

Chapter 18

"N" Standard Extension for User-Level Interrupts, Version 1.1

This is a placeholder for a more complete writeup of the N extension, and to form a basis for discussion.

This chapter presents a proposal for adding RISC-V user-level interrupt and exception handling. When the N extension is present, and the outer execution environment has delegated designated interrupts and exceptions to user-level, then hardware can transfer control directly to a user-level trap handler without invoking the outer execution environment.

User-level interrupts are primarily intended to support secure embedded systems with only M-mode and U-mode present, but can also be supported in systems running Unix-like operating systems to support user-level trap handling.

When used in an Unix environment, the user-level interrupts would likely not replace conventional signal handling, but could be used as a building block for further extensions that generate user-level events such as garbage collection barriers, integer overflow, floating-point traps.

18.1 Additional CSRs

The user-visible CSRs added to support the N extension are listed in Table 18.1.

Number	Name	Description
0x000	ustatus	User status register.
0x004	uie	User interrupt-enable register.
0x005	utvec	User trap handler base address.
0x040	uscratch	Scratch register for user trap handlers.
0x041	uepc	User exception program counter.
0x042	ucause	User trap cause.
0x043	utval	User bad address or instruction.
0x044	uip	User interrupt pending.

Table 18.1: CSRs for N extension.

18.2 User Status Register (ustatus)

The ustatus register is an XLEN-bit read/write register formatted as shown in Figure 18.1. The ustatus register keeps track of and controls the hart's current operating state.

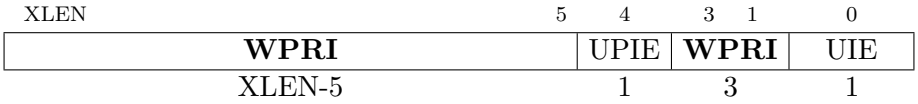


Figure 18.1: User-mode status register (ustatus).

The user interrupt-enable bit UIE disables user-level interrupts when clear. The value of UIE is copied into UPIE when a user-level trap is taken, and the value of UIE is set to zero to provide atomicity for the user-level trap handler.

There is no UPP bit to hold the previous privilege mode as it can only be user mode.

The URET instructions is used to return from traps in U-mode, and URET copies UPIE into UIE, then sets UPIE.

UPIE is set after the UPIE/UIE stack is popped to enable interrupts and help catch coding errors.

18.3 Other CSRs

The remaining CSRs function in an analogous way to the trap handling registers defined for M-mode and S-mode.

A more complete writeup to follow.

18.4 N Extension Instructions

The URET instruction is added to perform the analogous function to MRET and SRET.

18.5 Reducing Context-Swap Overhead

The user-level interrupt-handling registers add considerable state to the user-level context, yet will usually rarely be active in normal use. In particular, uepc, ucause, and utval are only valid during execution of a trap handler.

An NS field can be added to mstatus and sstatus following the format of the FS and XS fields to reduce context-switch overhead when the values are not live. Execution of URET will place the uepc, ucause, and utval back into initial state.

Chapter 19

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) as a "general-purpose" ISA, and we use the abbreviation G for the IMAFD 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]								($> 32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

Table 19.1: RISC-V base opcode map, inst[1:0]=11

Table 19.1 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 12 can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

As we move beyond IMAFDC into further instruction set extensions, the added instructions tend to be more domain-specific and only provide benefits to a restricted class of applications, e.g., for multimedia or security. Unlike most commercial ISAs, the RISC-V ISA design clearly separates the base ISA and broadly applicable standard extensions from these more specialized additions. Chapter 21 has a more extensive discussion of ways to add extensions to the RISC-V ISA.

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

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred	succ	00000	000	00000	0001111	FENCE
0000 | 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

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

RV64I Base Instruction Set (in addition to RV32I)

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

RV32M Standard Extension

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

RV64M Standard Extension (in addition to RV32M)

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

RV32A Standard Extension

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

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

RV64A Standard Extension (in addition to RV32A)

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

RV32F Standard Extension

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

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

RV64F Standard Extension (in addition to RV32F)

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

RV32D Standard Extension

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

RV64D Standard Extension (in addition to RV32D)

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

Table 19.2: Instruction listing for RISC-V

Table 19.3 lists the CSRs that have currently been allocated CSR addresses. The timers, counters, and floating-point CSRs are the only CSRs defined in this specification.

Number	Privilege	Name	Description
Floating-Point Control and Status Registers			
0x001	Read/write	fflags	Floating-Point Accrued Exceptions.
0x002	Read/write	frm	Floating-Point Dynamic Rounding Mode.
0x003	Read/write	fcsr	Floating-Point Control and Status Register (frm + fflags).
Counters and Timers			
0xC00	Read-only	cycle	Cycle counter for RDCYCLE instruction.
0xC01	Read-only	time	Timer for RDTIME instruction.
0xC02	Read-only	instret	Instructions-retired counter for RDINSTRET instruction.
0xC80	Read-only	cycleh	Upper 32 bits of cycle, RV32I only.
0xC81	Read-only	timeh	Upper 32 bits of time, RV32I only.
0xC82	Read-only	instreth	Upper 32 bits of instret, RV32I only.

Table 19.3: RISC-V control and status register (CSR) address map.

Chapter 20

RISC-V Assembly Programmer's Handbook

This chapter is a placeholder for an assembly programmer's manual.

Table 20.1 lists the assembler mnemonics for the `x` and `f` registers and their role in the standard calling convention.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 20.1: Assembler mnemonics for RISC-V integer and floating-point registers.

Tables 20.2 and 20.3 contain a listing of standard RISC-V pseudoinstructions.

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Table 20.2: RISC-V pseudoinstructions.

Pseudoinstruction	Base Instruction	Meaning
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Table 20.3: Pseudoinstructions for accessing control and status registers.

Chapter 21

Extending RISC-V

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

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

21.1 Extension Terminology

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

Standard versus Non-Standard Extension

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

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

Instruction Encoding Spaces and Prefixes

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

We use the term *prefix* to refer to the bits to the *right* of an instruction encoding space (since RISC-V is little-endian, the bits to the right are stored at earlier memory addresses, hence form a prefix in instruction-fetch order). The prefix for the standard base ISA encoding is the two-bit "11" field held in bits 1–0 of the 32-bit word, while the prefix for the standard atomic extension "A" is the seven-bit "0101111" field held in bits 6–0 of the 32-bit word representing the AMO major opcode. A quirk of the encoding format is that the 3-bit funct3 field used to encode a minor opcode is not contiguous with the major opcode bits in the 32-bit instruction format, but is considered part of the prefix for 22-bit instruction spaces.

Although an instruction encoding space could be of any size, adopting a smaller set of common sizes simplifies packing independently developed extensions into a single global encoding. Table 21.1 gives the suggested sizes for RISC-V.

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

Table 21.1: Suggested standard RISC-V instruction encoding space sizes.

Greenfield versus Brownfield Extensions

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

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

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

Table 21.2: Two-dimensional characterization of standard instruction-set extensions.

Table 21.2 shows the bases and standard extensions placed in a simple two-dimensional taxonomy. One axis is whether the extension is greenfield or brownfield, while the other axis is whether the extension adds architectural state. For greenfield extensions, the size of the instruction encoding space is given in parentheses. For brownfield extensions, the name of the extension (greenfield or brownfield) it builds upon is given in parentheses. Additional user-level architectural state usually implies changes to the supervisor-level system or possibly to the standard calling convention.

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

Standard-Compatible Global Encodings

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

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

Guaranteed Non-Standard Encoding Space

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

21.2 RISC-V Extension Design Philosophy

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

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

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

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

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

21.3 Extensions within fixed-width 32-bit instruction format

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

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

Available 30-bit instruction encoding spaces

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

Available 25-bit instruction encoding spaces

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

There are four major opcodes expressly reserved for custom extensions (Table 19.1), each of which represents a 25-bit encoding space. Two of these are reserved for eventual use in the RV128 base encoding (will be OP-IMM-64 and OP-64), but can be used for standard or non-standard extensions for RV32 and RV64.

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

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

If an implementation does not require instructions longer than 32-bits, then an additional four major opcodes are available (those marked in gray in Table 19.1).

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

Available 22-bit instruction encoding spaces

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

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

Other spaces

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

21.4 Adding aligned 64-bit instruction extensions

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

To simplify use of standard tools, the 64-bit instructions should be encoded as described in Figure 1.1. However, an implementation might choose a non-standard instruction-length encoding for 64-bit instructions, while retaining the standard encoding for 32-bit instructions. For example, if compressed instructions are not required, then a 64-bit instruction could be encoded using one or more zero bits in the first two bits of an instruction.

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

21.5 Supporting VLIW encodings

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

Fixed-size instruction group

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

Encoded-Length Groups

Another approach is to use the standard length encoding from Figure 1.1 to encode parallel instruction groups, allowing NOPs to be compressed out of the VLIW instruction. For example, a 64-bit instruction could hold two 28-bit operations, while a 96-bit instruction could hold three 28-bit operations, and so on. Alternatively, a 48-bit instruction could hold one 42-bit operation, while a 96-bit instruction could hold two 42-bit operations, and so on.

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

Fixed-Size Instruction Bundles

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

End-of-Group bits in Prefix

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

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

Chapter 22

ISA Subset Naming Conventions

This chapter describes the RISC-V ISA subset naming scheme that is used to concisely describe the set of instructions present in a hardware implementation, or the set of instructions used by an application binary interface (ABI).

The RISC-V ISA is designed to support a wide variety of implementations with various experimental instruction-set extensions. We have found that an organized naming scheme simplifies software tools and documentation.

22.1 Case Sensitivity

The ISA naming strings are case insensitive.

22.2 Base Integer ISA

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

22.3 Instruction Extensions Names

Standard ISA extensions are given a name consisting of a single letter. For example, the first four standard extensions to the integer bases are: "M" for integer multiplication and division, "A" for atomic memory instructions, "F" for single-precision floating-point instructions, and "D" for double-precision floating-point instructions. Any RISC-V instruction set variant can be succinctly described by concatenating the base integer prefix with the names of the included extensions. For example, "RV64IMAFD".

We have also defined an abbreviation "G" to represent the "IMAFD" base and extensions, as this is intended to represent our standard general-purpose ISA.

Standard extensions to the RISC-V ISA are given other reserved letters, e.g., "Q" for quad-precision floating-point, or "C" for the 16-bit compressed instruction format.

22.4 Version Numbers

Recognizing that instruction sets may expand or alter over time, we encode subset version numbers following the subset name. Version numbers are divided into major and minor version numbers, separated by a "p". If the minor version is "0", then "p0" can be omitted from the version string. Changes in major version numbers imply a loss of backwards compatibility, whereas changes in only the minor version number must be backwards-compatible. For example, the original 64-bit standard ISA defined in release 1.0 of this manual can be written in full as "RV64I1p0M1p0A1p0F1p0D1p0", more concisely as "RV64I1M1A1F1D1", or even more concisely as "RV64G1". The G ISA subset can be written as "RV64I2p0M2p0A2p0F2p0D2p0", or more concisely "RV64G2".

We introduced the version numbering scheme with the second release, which we also intend to become a permanent standard. Hence, we define the default version of a standard subset to be that present at the time of this document, e.g., "RV32G" is equivalent to "RV32I2M2A2F2D2".

22.5 Non-Standard Extension Names

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

Non-standard extensions must be separated from other multi-letter extensions by a single underscore. For example, an ISA with non-standard extensions Argle and Bargle may be named "RV64GXargle_Xbargle".

22.6 Supervisor-level Instruction Subsets

Standard supervisor instruction subsets are defined in Volume II, but are named using "S" as a prefix, followed by a supervisor subset name beginning with a letter and an optional version number.

Supervisor extensions must be separated from other multi-letter extensions by a single underscore.

22.7 Supervisor-level Extensions

Non-standard extensions to the supervisor-level ISA are defined using the "SX" prefix.

22.8 Subset Naming Convention

Table 22.1 summarizes the standardized subset names.

Subset	Name
Standard General-Purpose ISA	
Integer	I
Integer Multiplication and Division	M
Atomics	A
Single-Precision Floating-Point	F
Double-Precision Floating-Point	D
General	G = IMAFD
Standard User-Level Extensions	
Quad-Precision Floating-Point	Q
Decimal Floating-Point	L
16-bit Compressed Instructions	C
Bit Manipulation	B
Dynamic Languages	J
Transactional Memory	T
Packed-SIMD Extensions	P
Vector Extensions	V
User-Level Interrupts	N
Non-Standard User-Level Extensions	
Non-standard extension "abc"	Xabc
Standard Supervisor-Level ISA	
Supervisor extension "def"	Sdef
Non-Standard Supervisor-Level Extensions	
Supervisor extension "ghi"	SXghi

Table 22.1: Standard ISA subset names. The table also defines the canonical order in which subset names must appear in the name string, with top-to-bottom in table indicating first-to-last in the name string, e.g., RV32IMAFDQC is legal, whereas RV32IMAFDCQ is not.

Chapter 23

History and Acknowledgments

23.1 History from Revision 1.0 of ISA manual

The RISC-V ISA and instruction set manual builds upon several earlier projects. Several aspects of the supervisor-level machine and the overall format of the manual date back to the T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. T0 was a vector processor based on the MIPS-II ISA, with Krste Asanović as main architect and RTL designer, and Brian Kingsbury and Bertrand Irrisou as principal VLSI implementors. David Johnson at ICSI was a major contributor to the T0 ISA design, particularly supervisor mode, and to the manual text. John Hauser also provided considerable feedback on the T0 ISA design.

The Scale (Software-Controlled Architecture for Low Energy) project at MIT, begun in 2000, built upon the T0 project infrastructure, refined the supervisor-level interface, and moved away from the MIPS scalar ISA by dropping the branch delay slot. Ronny Krashinsky and Christopher Batten were the principal architects of the Scale Vector-Thread processor at MIT, while Mark Hampton ported the GCC-based compiler infrastructure and tools for Scale.

A lightly edited version of the T0 MIPS scalar processor specification (MIPS-6371) was used in teaching a new version of the MIT 6.371 Introduction to VLSI Systems class in the Fall 2002 semester, with Chris Terman and Krste Asanović as lecturers. Chris Terman contributed most of the lab material for the class (there was no TA!). The 6.371 class evolved into the trial 6.884 Complex Digital Design class at MIT, taught by Arvind and Krste Asanović in Spring 2005, which became a regular Spring class 6.375. A reduced version of the Scale MIPS-based scalar ISA, named SMIPS, was used in 6.884/6.375. Christopher Batten was the TA for the early offerings of these classes and developed a considerable amount of documentation and lab material based around the SMIPS ISA. This same SMIPS lab material was adapted and enhanced by TA Yunsup Lee for the UC Berkeley Fall 2009 CS250 VLSI Systems Design class taught by John Wawrzynek, Krste Asanović, and John Lazzaro.

The Maven (Malleable Array of Vector-thread ENgines) project was a second-generation vector-thread architecture. Its design was led by Christopher Batten when he was an Exchange Scholar at UC Berkeley starting in summer 2007. Hidetaka Aoki, a visiting industrial fellow from Hitachi, gave considerable feedback on the early Maven ISA and microarchitecture design. The Maven

infrastructure was based on the Scale infrastructure but the Maven ISA moved further away from the MIPS ISA variant defined in Scale, with a unified floating-point and integer register file. Maven was designed to support experimentation with alternative data-parallel accelerators. Yunsup Lee was the main implementor of the various Maven vector units, while Rimas Avižienis was the main implementor of the various Maven scalar units. Yunsup Lee and Christopher Batten ported GCC to work with the new Maven ISA. Christopher Celio provided the initial definition of a traditional vector instruction set ("Flood") variant of Maven.

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

23.2 History from Revision 2.0 of ISA manual

Multiple implementations of RISC-V processors have been completed, including several silicon fabrications, as shown in Figure 23.1.

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

Table 23.1: Fabricated RISC-V testchips.

The first RISC-V processors to be fabricated were written in Verilog and manufactured in a pre-production 28 nm FDSOI technology from ST as the Raven-1 testchip in 2011. Two cores were developed by Yunsup Lee and Andrew Waterman, advised by Krste Asanović, and fabricated together: 1) an RV64 scalar core with error-detecting flip-flops, and 2) an RV64 core with an attached 64-bit floating-point vector unit. The first microarchitecture was informally known as "TrainWreck", due to the short time available to complete the design with immature design libraries.

Subsequently, a clean microarchitecture for an in-order decoupled RV64 core was developed by Andrew Waterman, Rimas Avižienis, and Yunsup Lee, advised by Krste Asanović, and, continuing the railway theme, was codenamed "Rocket" after George Stephenson's successful steam locomotive design. Rocket was written in Chisel, a new hardware design language developed at UC Berkeley. The IEEE floating-point units used in Rocket were developed by John Hauser, Andrew Waterman, and Brian Richards. Rocket has since been refined and developed further, and has been fabricated two more times in 28 nm FDSOI (Raven-2, Raven-3), and five times in IBM 45 nm SOI technology

(EOS14, EOS16, EOS18, EOS20, EOS22) for a photonics project. Work is ongoing to make the Rocket design available as a parameterized RISC-V processor generator.

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

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

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

Andrew Waterman and Yunsup Lee developed the C++ ISA simulator "Spike", used as a golden model in development and named after the golden spike used to celebrate completion of the US transcontinental railway. Spike has been made available as a BSD open-source project.

Andrew Waterman completed a Master's thesis with a preliminary design of the RISC-V compressed instruction set [33].

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

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

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

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

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

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

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

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

Acknowledgments

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

23.3 History from Revision 2.1

Uptake of the RISC-V ISA has been very rapid since the introduction of the frozen version 2.0 in May 2014, with too much activity to record in a short history section such as this. Perhaps the most important single event was the formation of the non-profit RISC-V Foundation in August 2015. The Foundation will now take over stewardship of the official RISC-V ISA standard, and the official website `riscv.org` is the best place to obtain news and updates on the RISC-V standard.

Acknowledgments

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

23.4 History from Revision 2.2

Acknowledgments

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

23.5 History for Revision 2.3

Uptake of RISC-V continues at breakneck pace.

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

Daniel Lustig, Arvid, and [**FIXME:**] contributed the RISC-V memory consistency model.

23.6 Funding

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

- **Par Lab:** Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.
- **Project Isis:** DoE Award DE-SC0003624.
- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. DARPA POEM program Award HR0011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Hewlett Packard Enterprise, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

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

Appendix A

RVWMO Explanatory Material

This section provides more explanation for the RVWMO memory model, using more informal language and concrete examples. These are intended to clarify the meaning and intent of the axioms and preserved program order rules.

A.1 Why RVWMO?

Memory consistency models fall along a loose spectrum from weak to strong. Weak memory models (e.g., ARMv7, Power, Alpha) allow more hardware implementation flexibility and deliver arguably better performance, performance per watt, power, scalability, and hardware verification overheads than strong models, at the expense of a more complex programming model. Strong models (e.g., sequential consistency, TSO) provide simpler programming models, but at the cost of imposing more restrictions on the kinds of hardware optimizations that can be performed in the pipeline and in the memory system, with some cost to power and area overheads, and with some added hardware verification burden.

For the base ISA, RISC-V has chosen the RVWMO memory model, which is a variant of release consistency. This places it in between the two extremes of the memory model spectrum. It is not as weak as the Power memory model, and this buys back some programming model simplicity without giving up very much in terms of performance. RVWMO is also not as restrictive as RVTSO, and hence it remains weak enough to ensure that implementations can be performant and scalable without incurring huge hardware complexity overheads. RVWMO is similar to the ARMv8 memory model in this regard.

As such, the RVWMO memory model enables architects to build simple implementations, aggressive implementations, implementations embedded deeply inside a much larger system and subject to complex memory system interactions, or any number of other possibilities, all while simultaneously being strong enough to support programming language memory models at high performance.

The risk of a weak memory model lies in the complexity of the programming model. Buggy code which "just worked" on stronger implementations may well break on more aggressive implementations due to the bugs simply not manifesting on the stronger-than-necessary implementations. For

these situations, though, the root cause is the bug in the original software, not the memory model itself. The risk of finding short-term bugs in code ported from other architectures is outweighed by the long-term benefits that the weak memory model delivers more generally.

To mitigate this risk, some hardware implementations may choose to stick with RVTSO, and that is perfectly acceptable and perfectly compatible with the RVWMO memory model. The cost that the weak memory model imposes on such implementations is the incremental overhead of fetching instructions (e.g., `fence r,rw` and `fence rw,w`) which become no-ops on that implementation. (These fences must remain present in the code to ensure compatibility with other more weakly-ordered RISC-V implementations.)

Most software is also fully compatible with weak memory models. C/C++, Java, and Linux, to name some of the most notable and more formally analyzed examples, are all entirely compatible with weak non-atomic memory models, as all are designed to run not just on x86 but also on ARM, Power, and many other architectures. It is true that some code, e.g., code ported from x86, does sometimes (correctly or incorrectly) assume a stronger model such as TSO. For such code, the RVWMO memory model provides a means for restoring TSO to sections of code through fences and atomics with `.aq` and `.rl` bits in the "A" extension, until such code can be ported to RVWMO over time.

Designers who wish to provide drop-in compatibility with x86 code can also implement the Ztso extension which enforces RVTSO. Code written for RVWMO is automatically and inherently compatible with RVTSO, but code written assuming RVTSO is not guaranteed to run correctly on RVWMO implementations. In fact, RVWMO implementations will (and should) simply refuse to run TSO-only binaries. Each implementation must therefore choose whether to prioritize compatibility with RVTSO code (e.g., to facilitate porting from x86) or whether to instead prioritize compatibility with other RISC-V cores implementing RVWMO.

A.2 Litmus Tests

The explanations in this chapter make use of *litmus tests*, or small programs designed to test or highlight one particular aspect of a memory model. Figure A.1 shows an example of a litmus test with two harts. For this figure (and for all figures that follow in this chapter), we assume that `s0`–`s2` are pre-set to the same value in all harts. As a convention, we will assume that `s0` holds the address labeled `x`, `s1` holds `y`, and `s2` holds `z`, where `x`, `y`, and `z` are different memory addresses. This figure shows the same program twice: on the left in RISC-V assembly, and again on the right in graphical form.

Litmus tests are used to understand the implications of the memory model in specific concrete situations. For example, in the litmus test of Figure A.1, the final value of `a0` in the first hart can be either 2, 4, or 5, depending on the dynamic interleaving of the instruction stream from each hart at runtime. However, in this example, the final value of `a0` in Hart 0 will never be 1 or 3: the value 1 will no longer be visible at the time the load executes, and the value 3 will not yet be visible by the time the load executes.

We analyze this test and many others below.

```
      Hart 0                 Hart 1
      ⋮                      ⋮
      li t1, 1               li t4, 4
(a)   sw t1,0(s0)      (e)   sw t4,0(s0)
      ⋮                      ⋮
      li t2, 2
(b)   sw t2,0(s0)
      ⋮                      ⋮
(c)   lw a0,0(s0)
      ⋮                      ⋮
      li t3, 3               li t5, 5
(d)   sw t3,0(s0)      (f)   sw t5,0(s0)
      ⋮                      ⋮
```

(picture coming soon)

Figure A.1: A sample litmus test

A.3 Explaining the RVWMO Rules

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

A.3.1 Preserved Program Order and Global Memory Order

Preserved program order represents the set of intra-hart orderings that the hart's pipeline must ensure are maintained as the instructions execute, even in the presence of hardware optimizations that might otherwise reorder those operations. Events from the same hart which are not ordered by preserved program order, on the other hand, may appear reordered from the perspective of other harts and/or observers.

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

The order in which loads perform does not always directly correspond to the relative age of the values those two loads return. In particular, a load b may perform before another load a to the same address (i.e., b may execute before a, and b may appear before a in the global memory order), but a may nevertheless return an older value than b. This discrepancy captures the reordering effects of store buffers placed between the core and memory: a younger load may read from a value in the store buffer, while an older load which appears before that store in program order may ignore that younger store and read an older value from memory instead. To account for this, at the time each load performs, the value it returns is determined by the load value axiom, not just strictly by

determining the most recent store to the same address in the global memory order, as described below.

A.3.2 Store Buffering (Load Value Axiom)

> Load value axiom: Each byte of each load returns the corresponding byte written by the whichever of the following two stores comes later in the global memory order:
>
> 1. the latest store to the same address and preceding the load in the global memory order
> 2. the latest store to the same address and preceding the load in program order

Preserved program order is *not* required to respect the ordering of a store followed by a load to an overlapping address. This complexity arises due to the ubiquity of store buffers in nearly all implementations. Informally, the load may perform (return a value) by forwarding from the store while the store is still in the store buffer, and hence before the store itself performs (writes back to globally visible memory). Any other hart will therefore observe the load as performing before the store.

	Hart 0		Hart 1
	`li t1, 1`		`li t1, 1`
(a)	`sw t1,0(s0)`	(e)	`sw t1,0(s1)`
(b)	`lw a0,0(s0)`	(f)	`lw a2,0(s1)`
(c)	`fence r,r`	(g)	`fence r,r`
(d)	`lw a1,0(s1)`	(h)	`lw a3,0(s0)`

(picture coming soon)

Figure A.2: A store buffer forwarding litmus test

Consider the litmus test of Figure A.2. When running this program on an implementation with store buffers, it is possible to arrive at the final outcome `a0=1, a1=0, a2=1, a3=0` as follows:

- (a) executes and enters the first hart's private store buffer
- (b) executes and forwards its return value 1 from (a) in the store buffer
- (c) executes since all previous loads (i.e., (b)) have completed
- (d) executes and reads the value 0 from memory
- (e) executes and enters the second hart's private store buffer
- (f) executes and forwards its return value 1 from (d) in the store buffer
- (g) executes since all previous loads (i.e., (f)) have completed
- (h) executes and reads the value 0 from memory
- (a) drains from the first hart's store buffer to memory

- (e) drains from the second hart's store buffer to memory

Therefore, the memory model must be able to account for this behavior.

To put it another way, suppose the definition of preserved program order did include the following hypothetical rule: memory access a precedes memory access b in preserved program order (and hence also in the global memory order) if a precedes b in program order and a and b are accesses to the same memory location, a is a write, and b is a read. Call this "Rule X". Then we get the following:

- (a) precedes (b): by rule X
- (b) precedes (d): by rule 2
- (d) precedes (e): by the load value axiom. Otherwise, if (e) preceded (d), then (d) would be required to return the value 1. (This is a perfectly legal execution; it's just not the one in question)
- (e) precedes (f): by rule X
- (f) precedes (h): by rule 2
- (h) precedes (a): by the load value axiom, as above.

The global memory order must be a total order and cannot be cyclic, because a cycle would imply that every event in the cycle happens before itself, which is impossible. Therefore, the execution proposed above would be forbidden, and hence the addition of rule X would break the memory model.

Nevertheless, even if (b) precedes (a) and/or (f) precedes (e) in the global memory order, the only sensible possibility in this example is for (b) to return the value written by (a), and likewise for (f) and (e). This combination of circumstances is what leads to the second option in the definition of the load value axiom. Even though (b) precedes (a) in the global memory order, (a) will still be visible to (b) by virtue of sitting in the store buffer at the time (b) executes. Therefore, even if (b) precedes (a) in the global memory order, (b) should return the value written by (a) because (a) precedes (b) in program order. Likewise for (e) and (f).

A.3.3 Same-Address Orderings, Part 1 (Rule 1)

> Rule 1: a and b are accesses to overlapping memory addresses, and b is a store

Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model.

Same-address load-load orderings are far more subtle; see Chapter A.3.7.

A.3.4 Fences (Rule 2)

> Rule 2: a and b are separated in program order by a fence, a is in the predecessor set of the fence, and b is in the successor set of the fence

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

The full RISC-V opcode encoding currently has nine non-trivial combinations of the four bits `pr`, `pw`, `sr`, and `sw`, plus one extra encoding which is expected to be added to facilitate mapping of "acquire+release" or TSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, only six are commonly used in practice:

- `fence rw,rw`
- `fence.tso` (i.e., a combined `fence r,rw` + `fence rw,w`)
- `fence rw,w`
- `fence r,rw`
- `fence r,r`
- `fence w,w`

We strongly recommend that programmers stick to these six, as these are the best understood. `fence` instructions using any other combination of `.pr`, `.pw`, `.sr`, and `.sw` are reserved.

Finally, we note that since RISC-V uses a multi-copy atomic memory model, programmers can reason about fences and the `.aq` and `.rl` bits in a thread-local manner. There is no complex notion of "fence cumulativity" as found in memory models which are not multi-copy atomic.

A.3.5 Acquire/Release Ordering (Rules 3–7)

> Rule 3: a is a load-acquire
> Rule 4: b is a store-release
> Rule 5: a is a store-release-RCsc and b is a load-acquire-RCsc
> Rule 6: a is a store-SC
> Rule 7: b is a load-SC

An *acquire* operation is used at the start of a critical section. The general requirement for acquire semantics is that all loads and stores inside the critical section are up to date with respect to the

synchronization variable being used to protect it. In other words, an acquire operation requires load-to-load/store ordering. Acquire ordering can be enforced in one of two ways: setting `.aq`, which enforces ordering with respect to just the synchronization variable itself, or with a `FENCE r,rw`, which enforces ordering with respect to all previous loads.

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

Figure A.3: A spinlock with atomics

Consider Figure A.3: Because this example uses `.aq`, the loads and stores in the critical section are guaranteed to appear in the global memory order after the `amoswap` used to acquire the lock. However, assuming `a0`, `a1`, and `a2` point to different memory locations, the loads and stores in the critical section may or may not appear after the "random unrelated load" at the beginning of the example in the global memory order.

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

Figure A.4: A spinlock with fences

Now, consider the alternative in Figure A.4. In this case, even though the `amoswap` does not enforce ordering with an `.aq` bit, the fence nevertheless enforces that the acquire `amoswap` appears earlier in the global memory order than all loads and stores in the critical section. Note, however, that in this case, the fence also enforces additional orderings: it also requires that the "random unrelated load" at the start of the program appears also appears earlier in the global memory order than the loads and stores of the critical section. (This particular fence does not, however, enforce any

ordering with respect to the "random unrelated store" at the start of the snippet.) In this way, fence-enforced orderings are slightly coarser than orderings enforced by `.aq`.

Release orderings work exactly the same as acquire orderings, just in the opposite direction. Release semantics require all loads and stores in the critical section to appear before the lock-releasing store (here, an `amoswap`) in the global memory order. This can be enforced using the `.rl` bit or with a `fence rw,w` operations. Likewise, the ordering between the loads and stores in the critical section and the "random unrelated store" at the end of the code snippet is enforced only by the `fence rw,w` in the second example, not by the `.rl` in the first example.

By default, store-release-to-load-acquire ordering is not enforced. This facilitates the porting of code written under the TSO and/or RCpc memory models; see Chapter A.6 for details. To enforce store-release-to-load-acquire ordering, use store-release-RCsc and load-acquire-RCsc operations, so that PPO rule 5 applies. The use of only store-release-RCsc and load-acquire-RCsc operations implies sequential consistency, as the combination of PPO rules 3–5 implies that all RCsc accesses will respect program order.

AMOs with both `.aq` and `.rl` set are fully-ordered operations. Treating the load part and the store part as independent RCsc operations is not in and of itself sufficient to enforce full fencing behavior, but this subtle weak behavior is counterintuitive and not much of an advantage architecturally, especially with `lr` and `sc` also available. For this reason, AMOs annotated with `.aqrl` are strengthened to being fully-ordered under RVWMO.

A.3.6 Dependencies (Rules 8–11)

> Rule 8: a is a load, and b has a syntactic address dependency on a
> Rule 9: a is a load, b is a store, and b has a syntactic data dependency on a
> Rule 10: a is a load, b is a store, and b has a syntactic control dependency on a
> Rule 11: a is a load, and there exists some m such that m has an address or data dependency on a and b has a success dependency on m

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

Like other modern memory models, the RVWMO memory model uses syntactic rather than semantic dependencies. In other words, this definition depends on the identities of the registers being accessed by different instructions, not the actual contents of those registers. This means that an address, control, or data dependency must be enforced even if the calculation could seemingly be "optimized away". This choice ensures that RVWMO remains compatible with programmers that use these false syntactic dependencies intentionally to form a lightweight type of ordering mechanism.

For example, there is a syntactic address dependency from the first instruction to the last instruction in the Figure A.5, even though `a1` XOR `a1` is zero and hence has no effect on the address accessed by the second load.

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

Figure A.5: A syntactic address dependency

The benefit of using dependencies as a lightweight synchronization mechanism is that the ordering enforcement requirement is limited only to the specific two instructions in question. Other non-dependent instructions may be freely-reordered by aggressive implementations. One alternative would be to use a load-acquire, but this would enforce ordering for the first load with respect to *all* subsequent instructions. Another would be to use a `fence r,r`, but this would include all previous and all subsequent loads, making this option each more expensive.

Control dependencies behave differently from address and data dependencies in the sense that a control dependency always extends to all instructions following the original target in program order. Consider Figure A.6: the instruction at `next` will always execute, but it nevertheless still has control dependency from the first instruction.

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

Figure A.6: A syntactic control dependency

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

Figure A.7: Another syntactic control dependency

Likewise, consider Figure A.7. Even though both branch outcomes have the same target, there is still a control dependency from the first instruction in this snippet to the last. This definition of control dependency is subtly stronger than what might be seen in other contexts (e.g., C++), but it conforms with standard definitions of control dependencies in the literature.

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

(picture coming soon)

Figure A.8: A variant of the LB litmus test

Finally, we highlight a unique new rule regarding the success registers written by store-conditional instructions. In certain cases, without PPO rule 11, a store conditional could in theory be made to

store its own success output value as its data, in a manner reminiscent of so-called out-of-thin-air behavior. This is shown in Figure A.8. Suppose a hypothetical implementation could occasionally make some early guarantee that a store-conditional operation will succeed. In this case, (c) could return 0 to `a2` early (before actually executing), allowing the sequence (d), (e), (f), (a), and then (b) to execute, and then (c) might execute (successfully) only at that point. This would imply that (c) writes its own success value to `0(s1)`!

To rule out this bizarre behavior, PPO rule 11 says that store-conditional instructions may not return success or failure into the destination register until both the address and data for the instruction have been resolved. In the example above, this would enforce an ordering from (a) to (d), and this would in turn form a cycle that rules out the strange proposed execution.

A.3.7 Same-Address Load-Load Ordering (Rule 12)

> Rule 12: a and b are loads to overlapping memory addresses, there is no store to overlapping memory location(s) between a and b in program order, and a and b return values from different stores

In contrast to same-address orderings ending in a store, same-address load-load ordering requirements are very subtle.

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

	Hart 0		Hart 1
	`li t1, 1`		`li t2, 2`
(a)	`sw t1,0(s0)`	(d)	`lw a0,0(s1)`
(b)	`fence w, w`	(e)	`sw t2,0(s1)`
(c)	`sw t1,0(s1)`	(f)	`lw a1,0(s1)`
		(g)	`xor t3,a1,a1`
		(h)	`add s0,s0,t3`
		(i)	`lw a2,0(s0)`

(picture coming soon)

Figure A.9: Litmus test MP+FENCE+fri-rfi-addr

Consider the litmus test of Figure A.9, which is one particular instance of the more general "fri-rfi" pattern. The term "fri-rfi" refers to the sequence (d),(e),(f): (d) "from-reads" (i.e., reads from an earlier write than) (e) which is the same hart, and (f) reads from (e) which is in the same hart.

From a microarchitectural perspective, outcome `a0=1, a1=2, a2=0` is legal (as are various other less subtle outcomes). Intuitively, the following would produce the outcome in question:

- (a), (b), (c) execute
- (d) stalls (for whatever reason; perhaps it's stalled waiting for some other preceding instruction)
- (e) executes and enters the store buffer
- (f) forwards from (e) in the store buffer
- (g), (h), and (i) execute
- (d) unstalls and executes
- (e) drains from the store buffer to memory

This corresponds to a global memory order of (e),(f),(i),(a),(c),(d). Note that even though (f) performs before (d), the value returned by (f) is newer than the value returned by (d). Therefore, this execution is legal and does not violate the CoRR requirements even though (f) appears before (d) in global memory order.

Likewise, if two back-to-back loads return the values written by the same store, then they may also appear out-of-order in the global memory order without violating CoRR. Note that this is not the same as saying that the two loads return the same value, since two different stores may write the same value. Consider the litmus test of Figure A.10:

	Hart 0		Hart 1	
	`li t1, 1`	(d)	`lw a0,0(s1)`	
(a)	`sw t1,0(s0)`	(e)	`xor t2,a0,a0`	
(b)	`fence w, w`	(f)	`add s2,s2,t2`	
(c)	`sw t1,0(s1)`	(g)	`lw a1,0(s2)`	(picture coming soon)
		(h)	`lw a2,0(s2)`	
		(i)	`xor t3,a2,a2`	
		(j)	`add s0,s0,t3`	
		(k)	`lw a3,0(s0)`	

Figure A.10: Litmus test RSW

The outcome `a0=1,a1=a2,a3=0` can be observed by allowing (g) and (h) to be reordered. This might be done speculatively, and the speculation can be justified by the microarchitecture (e.g., by snooping for cache invalidations and finding none) because replaying (h) after (g) would return the value written by the same store anyway. Hence assuming `a1=a2`, (g) and (h) can be reordered. The global memory order corresponding to this execution would be (h),(k),(a),(c),(d),(g).

Executions of the above test in which `a1` does not equal `a2` do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in fact result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, PPO rule 12 forbids this CoRR violation from occurring. As such, PPO rule 12 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

A.3.8 Atomics and LR/SCs (Atomicity Axiom)

> Atomicity axiom: If r and w are a paired load and store, and if s is any store from which r returns a value, then there can be no store from another hart to an overlapping memory location which follows both r and s and which precedes w in the global memory order.

The RISC-V architecture decouples the notion of atomicity from the notion of ordering. Unlike architectures such as TSO, RISC-V atomics under RVWMO do not impose any ordering requirements by default. Ordering semantics are only guaranteed by the PPO rules that otherwise apply. This relaxed nature allows implementations to be aggressive about forwarding values even before a paired store has been committed to memory.

Roughly speaking, the atomicity rule states that there can be no store from another hart during the time the reservation is held. For AMOs, the reservation is held as the AMO is being performed. For successful `lr/sc` pairs, the reservation is held between the time the `lr` is performed and the time the `sc` is performed. In most cases, the atomicity rule states that there can be no store from another hart between the load and its paired store in global memory order.

There is one exception, however: if the paired load returns its value from a store s still in the store buffer (which some implementations may permit), then the reservation may not need to be acquired until s is ready to leave the store buffer, and this may occur after the paired load has already performed. Therefore, in this case, the requirement is only that no other store from another hart to an overlapping address can appear between time that s performs and the time that the paired store performs. Consider the example of Figure A.11:

Hart 0			Hart 1		
	li	t1, 2		li	t3, 2
	li	t2, 1		li	t4, 1
(a)	sd	t1,0(s0)	(d)	sd	t3,0(s1)
(b)	amoor.aq	a0,t2,0(s0)	(e)	amoswap.rl	x0,t4,0(s0)
(c)	sd	t2,0(s1)			

(picture coming soon)

Figure A.11: A litmus test where the reservation for `0(s0)` may not be acquired until after the load of (b) has already completed

The outcome `0(s0)=3, 0(s1)=2` is legal, with the global memory order of (b0),(c),(d),(e),(a),(b1), where (b0) and (b1) represent the load and store parts, respectively, of (b). The atomic operation (b) does not need to grab the reservation until (a) is ready to leave the store buffer. Therefore, although (e) is a store to the same address from another hart, and even though (e) lies between (b0) and (b1) in global memory order, this execution does not violate the atomicity axiom because (e) comes after (a) in global memory order.

```
(a) lr t0, 0(a0)
(b) sd t1, 0(a0)
(c) sc t2, 0(a0)
```

Figure A.12: Store-conditional (c) may succeed on some implementations

The atomicity rule does not forbid loads from being interleaved between the paired operations in program order or in the global memory order, nor does it forbid stores from the same hart

from appearing between the paired operations in either program order or in the global memory order. For example, the sequence in Figure A.12 is legal, and the `sc` may (but is not guaranteed to) succeed. By preserved program order rule 1, the program order of the three operations must be maintained in the global memory order. This does not violate the atomicity axiom, because the intervening non-conditional store is from the same hart as the paired load-reserved and store-conditional instructions.

```
        Hart 0                          Hart 1
    li t0, 1
(a) amoor.aq a0,t0,0(s0)  ||  (c) amoadd.aq a1,x0,0(s1)
(b) sd       a0,0(s1)     ||  (d) ld        a2,0(s0)
```

(picture coming soon)

Figure A.13: The `.aq` applies only to the load part of (a), and hence it does not order the store part of (a) before (b)

Likewise, in the test of Figure A.13, the following global memory order could result in the outcome `a1=1, a2=0`: (a0), (b), (c), (d), (a1).

Overall, the atomicity rule ensures that non-synchronization atomic operations (e.g., incrementing a counter) can be made as efficient as possible in high-performance implementations, while simultaneously ensuring that the atomicity conditions necessary for achieving consensus are maintained.

A.3.9 Pipeline Dependency Artifacts (Rules 13–16)

> Rule 13: a and b are loads, and there exists some store m between a and b in program order such that m has an address or data dependency on a, m accesses a memory address that overlaps the address accessed by b, and there is no other store to an overlapping memory location(s) between m and b in program order
> Rule 14: 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
> Rule 15: a and b are loads, b has a syntactic control dependency on a, and there exists a `fence.i` between the branch used to form the control dependency and b in program order
> Rule 16: a is a load, there exists an instruction m which has a syntactic address dependency on a, and there exists a `fence.i` between m and b in program order

These four "compound dependency" rules reflect behaviors of almost all real processor pipeline implementations, and they are added into the model explicitly to simplify the definition of the formal operational memory model and to improve compatibility with known patterns on other architectures.

```
(a)  lw a0, 0(s0)
(b)  sw a0, 0(s1)
(c)  lw a1, 0(s1)
```

(picture coming soon)

Figure A.14: Because of the data dependency from (a) to (b), (a) is also ordered before (c)

Rule 13 states that a load forward from a store until the address and data for that store are known.

Consider Figure A.14: (c) cannot be executed until the data for (b) has been resolved, because (c) must return the value written by (b) (or by something even later in the global memory order). Therefore, (c) will never execute before (a) has executed.

```
     li t1, 1
(a)  lw a0, 0(s0)
(b)  sw a0, 0(s1)
     sw t1, 0(s1)
(c)  lw a1, 0(s1)
```

(picture coming soon)

Figure A.15: Because of the extra store between (b) and (c), (a) is no longer necessarily ordered before (c)

If there were another store to the same address in between (b) and (c), as in Figure A.15, then (c) would no longer dependent on the data of (b) being resolved, and hence the dependency of (c) on (a), which produces the data for (b), would be broken.

One subtle related note is that `amoswap` does not contain a data dependency from its load to its store. Nor does every `sc` have a data dependency on its paired `lr`. Therefore, Rule 13 does not enforce an ordering from paired loads of this category to subsequent loads to overlapping addresses.

Rule 14 makes a similar observation to the previous rule: a store cannot be performed at memory until all previous loads which might access the same address have themselves been performed. Such a load must appear to execute before the store, but it cannot do so if the store were to overwrite the value in memory before the load had a chance to read the old value.

```
     li t1, 1
(a)  lw a0, 0(s0)
(b)  lw a1, 0(a0)
(c)  sw t1, 0(s1)
```

(picture coming soon)

Figure A.16: Because of the address dependency from (a) to (b), (a) is also ordered before (c)

Consider Figure A.16: (c) cannot be executed until the address for (b) is resolved, because it may turn out that the addresses match; i.e., that `a0=s1`. Therefore, (c) cannot be sent to memory before (a) has executed and confirmed whether the addresses to indeed overlap.

```
      (a)  ld a0, 0(s0)
           xor a1,a0,a0
           bne a1, critical
critical:  fence.i
      (c)  ld a1, 0(s1)
```

(picture coming soon)

Figure A.17: Because of the control dependency from (a) to (c), (a) is also ordered before (c)

Rule 15 reflects the idiom of Figure A.17 for a lightweight acquire fence: In this code snippet, (c) cannot execute until the `fence.i` is cleared. The `fence.i` cannot clear until the branch has executed and drained. The branch cannot execute until it receives the value from (a) through the `xor`. Therefore, (a) must be ordered before (c) in the global memory order.

Rule 16 and Figure A.18 present a similar situation: Once again, (c) cannot execute until the `fence.i` is cleared. The `fence.i` cannot clear until both (a) and (b) have at least issued (even if

```
(a)  ld a0, 0(s0)
(b)  ld a1, 0(a0)
     fence.i
(c)  ld a1, 0(s1)
```

(picture coming soon)

Figure A.18: Because of the address dependency from (a) to (b) and the `fence.i` between (b) and (c), (a) is also ordered before (c)

they have not yet returned a value). Finally, (b) cannot issue until it receives its address from (a). Therefore, (a) must be ordered before (c).

A.4 FENCE.I, SFENCE.VMA, and I/O Fences

In this section, we provide an informal description of how the `fence.i`, `sfence.vma`, and I/O fences interact with the memory model.

Instruction fetches and address translation operations (where applicable) follow the RISC-V memory model as well as the rules below.

- `fence.i`: Conceptually, `fence.i` ensures that no instructions following the `fence.i` are issued until all instructions prior to the `fence.i` have executed (but not necessarily performed globally). This implies that the fetch of each instruction following the `fence.i` in program order appears later in the global memory order than all stores prior to the `fence.i` in program order. That in turn means that instruction caches which hardware does not keep coherent with normal memory must be flushed when a `fence.i` instruction is executed. (`fence.i` is also used form the patterns of Chapter A.3.9.)

- `sfence.vma`: Conceptually, the instruction fetch and address translation operations of each instruction following the `sfence.vma` in program order appears later in the global memory order than all stores prior to the `sfence.vma` in program order. This implies that stale entries in the local hart's TLBs must be invalidated.

- Conceptually, updates to the page table made by a hardware page table walker form a paired atomic read-modify-write operation subject to the rules of the atomicity axiom

A.4.1 Coherence and Cacheability

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

- Main memory vs. I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.

- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Coherence and cacheability PMAs: neither the coherence nor the cacheability PMAs affect the memory model. The RISC-V privileged specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise. Likewise, non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless.
- Idempotency PMAs: Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

A.4.2 I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects. The preserved program order rules do not generally apply to I/O either. Instead, we informally say that memory access a is ordered before memory access b if a precedes b in program order and one or more of the following holds:

1. a and b are accesses to overlapping addresses in an I/O region
2. a and b are accesses to the same strongly-ordered I/O region
3. a and b are accesses to I/O regions, and the channel associated with the I/O region accessed by either a or b is channel 1
4. a and b are accesses to I/O regions associated with the same channel (except for channel 0)
5. a and b are separated in program order by a FENCE, a is in the predecessor set of the FENCE, and b is in the successor set of the FENCE. The predecessor and successor sets include the sets described by all eight FENCE bits `.pr`, `.pw`, `.pi`, `.po`, `.sr`, `.sw`, `.si`, and `.so`.
6. a and b are accesses to I/O regions, and a has `.aq` set
7. a and b are accesses to I/O regions, and b has `.rl` set
8. a and b are accesses to I/O regions, and a and b both have `.aq` and `.rl` set
9. a and b are accesses to I/O regions, and a is an AMO that has `.aq` and `.rl` set
10. a and b are accesses to I/O regions, and b is an AMO that has `.aq` and `.rl` set

As described above, accesses to I/O memory require stronger synchronization that what is enforced by the RVWMO PPO rules. For such cases, `FENCE` operations with `.pi`, `.po`, `.si`, and/or `.so` are needed. For example, to enforce ordering between a write to normal memory and an MMIO write to a device register, a `FENCE w,o` or stronger is needed. Even `.aq` and `.rl` do not enforce ordering

between normal memory accesses and accesses to I/O memory. When a fence is in fact used, implementations must assume that the device may attempt to access memory immediately after receiving the MMIO signal, and subsequent memory accesses from that device to memory must observe the effects of all accesses ordered prior to that MMIO operation.

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

Figure A.19: Ordering memory and I/O accesses

In other words, in Figure A.19, suppose `0(a0)` is in normal memory and `0(a1)` is the address of a device register in I/O memory. If the device accesses `0(a0)` upon receiving the MMIO write, then that load must conceptually appear after the first store to `0(a0)` according to the rules of the RVWMO memory model. In some implementations, the only way to ensure this will be to require that the first store does in fact complete before the MMIO write is issued. Other implementations may find ways to be more aggressive, while others still may not need to do anything different at all for I/O and normal memory accesses. Nevertheless, the RVWMO memory model does not distinguish between these options; it simply provides an implementation-agnostic mechanism to specify the orderings that must be enforced.

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

For implementations that conform to the RISC-V Unix Platform Specification, I/O devices, DMA operations, etc. are required to access memory coherently and via strongly-ordered I/O channels. Therefore, accesses to normal memory regions that are shared with I/O devices can also use the standard synchronization mechanisms. Implementations which do not conform to the Unix Platform Specification and/or in which devices do not access memory coherently will need to use platform-specific mechanisms (such as cache flushes) to enforce coherency.

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

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

A.5 Code Examples

A.5.1 Compare and Swap

An example using `lr/sc` to implement a compare-and-swap function is shown in Figure A.20. If inlined, compare-and-swap functionality need only take three instructions.

```
        # 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 a0, a2, (a0)    # Try to update.
        jr ra                # Return.
    fail:
        li a0, 1             # Set return to failure.
        jr ra                # Return.
```

Figure A.20: Sample code for compare-and-swap function using `lr/sc`.

A.5.2 Spinlocks

An example code sequence for a critical section guarded by a test-and-set spinlock is shown in Figure A.21. 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:
        amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
        bnez         t0, again    # Retry if held.
        # ...
        # Critical section.
        # ...
        amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Figure A.21: Sample code for mutual exclusion. `a0` contains the address of the lock.

A.6 Code Porting Guidelines

Normal x86 loads and stores are all inherently acquire and release operations: TSO enforces all load-load, load-store, and store-store ordering by default. All TSO loads must be mapped

onto `l{b|h|w|d}; fence r,rw`, and all TSO stores must either be mapped onto `amoswap.rl x0` or onto `fence rw,w; s{b|h|w|d}`. Alternatively, TSO loads and stores can be mapped onto `l{b|h|w|d}.aq` and `s{b|h|w|d}.rl` assembler pseudoinstructions to facilitate forwards compatibility in case such instructions are added to the ISA one day. However, in the meantime, the assembler will generate the same fence-based and/or `amoswap`-based versions for these pseudoinstructions. x86 atomics using the LOCK prefix are all sequentially consistent and when ported naively to RISC-V must be marked as `.aqrl`.

A Power `sync`/`hwsync` fence, an ARM `dmb` fence, and an x86 `mfence` are all equivalent to a RISC-V `fence rw,rw`. Power `isync` and ARM `isb` map to RISC-V `fence.i`. A Power `lwsync` map onto `fence.tso`, or onto `fence rw,rw` when `fence.tso` is not available. ARM `dmb ld` and `dmb st` fences map to RISC-V `fence r,rw` and `fence w,w`, respectively.

A direct mapping of ARMv8 atomics that maps unordered instructions to unordered instructions, RCpc instructions to RCpc instructions, and RCsc instructions to RCsc instructions is likely to work in the majority of cases. Mapping even unordered load-reserved instructions onto `lr.aq` (particularly for LR/SC pairs without internal data dependencies) is an even safer bet, as this ensures C/C++ release sequences will be respected. However, due to a subtle mismatch between the two models, strict theoretical compatibility with the ARMv8 memory model requires that a naive mapping translate all ARMv8 store conditional and load-acquire operations map onto RISC-V RCsc operations. Any atomics which are naively ported into RCsc operations may revert back to the straightforward mapping if the programmer can verify that the code is not relying on an ordering from the store-conditional to the load-acquire (as this is not common).

The Linux fences `smp_mb()`, `smp_wmb()`, `smp_rmb()` map onto `fence rw,rw`, `fence w,w`, and `fence r,r`, respectively. The fence `smp_read_barrier_depends()` map to a no-op due to preserved program order rules 8–10. The Linux fences `dma_rmb()` and `dma_wmb()` map onto `fence r,r` and `fence w,w`, respectively, since the RISC-V Unix Platform requires coherent DMA. The Linux fences `rmb()`, `wmb()`, and `mb()` map onto `fence ri,ri`, `fence wo,wo`, and `fence rwio,rwio`, respectively.

C/C++ Construct	RVWMO Mapping
Non-atomic load	`l{b|h|w|d}`
`atomic_load(memory_order_relaxed)`	`l{b|h|w|d}`
`atomic_load(memory_order_acquire)`	`l{b|h|w|d}; fence r,rw`
`atomic_load(memory_order_seq_cst)`	`fence rw,rw; l{b|h|w|d}; fence r,rw`
Non-atomic store	`s{b|h|w|d}`
`atomic_store(memory_order_relaxed)`	`s{b|h|w|d}`
`atomic_store(memory_order_release)`	`fence rw,w; s{b|h|w|d}`
`atomic_store(memory_order_seq_cst)`	`fence rw,rw; s{b|h|w|d}`
`atomic_thread_fence(memory_order_acquire)`	`fence r,rw`
`atomic_thread_fence(memory_order_release)`	`fence rw,w`
`atomic_thread_fence(memory_order_acq_rel)`	`fence.tso`
`atomic_thread_fence(memory_order_seq_cst)`	`fence rw,rw`

Table A.1: Mappings from C/C++ primitives to RISC-V primitives.

The C11/C++11 `memory_order_*` primitives should be mapped as shown in Table A.1. The `memory_order_acquire` orderings in particular must use fences rather than atomics to ensure that

release sequences behave correctly even in the presence of `amoswap`. The `memory_order_release` mappings may use `.rl` as an alternative.

Ordering Annotation	Fence-based Equivalent
`l{b|h|w|d|r}.aq`	`l{b|h|w|d|r}; fence r,rw`
`l{b|h|w|d|r}.aqrl`	`fence rw,rw; l{b|h|w|d|r}; fence r,rw`
`s{b|h|w|d|c}.rl`	`fence rw,w; s{b|h|w|d|c}`
`s{b|h|w|d|c}.aqrl`	`fence rw,w; s{b|h|w|d|c}`
`amo<op>.aq`	`amo<op>; fence r,rw`
`amo<op>.rl`	`fence rw,w; amo<op>`
`amo<op>.aqrl`	`fence rw,rw; amo<op>; fence rw,rw`

Table A.2: Mappings from `.aq` and/or `.rl` to fence-based equivalents. An alternative mapping places a `fence rw,rw` after the existing `s{b|h|w|d|c}` mapping rather than at the front of the `l{b|h|w|d|r}` mapping.

It is also safe to translate any `.aq`, `.rl`, or `.aqrl` annotation into the fence-based snippets of Table A.2. These can also be used as a legal implementation of `l{b|h|w|d}` or `s{b|h|w|d}` pseudoinstructions for as long as those instructions are not added to the ISA.

A.7 Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines below on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or "other-multi-copy-atomic"): any store value which is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no other early intra-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer, unless those two harts are the only two in the system. Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these

rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1–7 regularly and actively.
- expert programmers will use PPO rules 8–10 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 11–16 directly. These are included to facilitate common microarchitectural optimizations (rule 12) and the operational formal modeling approach (rules 11 and 13–16) described in Chapter **??**. They also facilitate the process of porting code from other architectures which have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1–4 and 6–7 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 5 may not be immediately obvious to architects, but is somewhat standard nevertheless
- The load value axiom, the atomicity axiom, and PPO rules 8–10 and 13–16 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not "optimized away".
- PPO rule 11 is not obvious, but it is necessary to avoid certain out-of-thin-air-like behavior that appears with store-conditional success values
- PPO rule 12 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were `fence rw,rw` (or `fence iorw,iorw`, if I/O is involved), regardless of the bits actually set
- implement all fences with `.pw` and `.sr` as if they were `fence rw,rw` (or `fence iorw,iorw`, if I/O is involved), as "`w,r`" is the most expensive of the four possible normal memory orderings anyway
- ignore any addresses passed to a fence instruction and simply implement the fence for all addresses
- implement an instruction with `.aq` set as being preceded immediately by `fence r,rw`

- implement an instruction with `.rl` set as being succeeded immediately by `fence rw,w`
- enforcing all same-address load-load ordering, even in the presence of patterns such as "fri-rfi" and "RSW"
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or `lr` to the same address
- forbid any forwarding of a value from an AMO or `sc` in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any normal memory fences that do not include "`w,r`" ordering
- implement all atomics to be RCsc; i.e., always enforce all store-release-to-load-acquire ordering

Architectures which implement RVTSO can safely do the following:

- Ignore all `.aq` and `.rl` bits, since these are implicitly always set under RVTSO. (`.aqrl` cannot be ignored, however, due to PPO rules 5–7.)
- Ignore all fences which do not have both `.pw` and `.sr` (unless the fence also orders I/O)
- Ignore PPO rules 1 and 8–16, since these are redundant with other PPO rules under RVTSO assumptions

Other general notes:

- Silent stores (i.e., stores which write the same value that already exists at a memory location) do not have any special behavior from a memory model point of view. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in cases such as RSW (Chapter A.3.7) which tend to be incompatible with silent stores.
- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

	Hart 0		Hart 1
	`li t1, 3`		`li t3, 2`
	`li t2, 1`		
(a)	`sw t1,0(s0)`	(d)	`lw a0,0(s1)`
(b)	`fence w, w`	(e)	`sw a0,0(s0)`
(c)	`sw t2,0(s1)`	(f)	`lw t3,0(s0)`

(picture coming soon)

Figure A.22: Write subsumption litmus test

As written, (a) must follow (f) in the global memory order:

- (a) follows (c) in the global memory order because of rule 2
- (c) follows (d) in the global memory order because of the Load Value axiom
- (d) follows (e) in the global memory order because of rule 7
- (e) follows (f) in the global memory order because of rule 1

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

A.8 Summary of New/Modified ISA Features

At a high level, PPO rules 5, 11, and 13–16 are all new rules that did not exist in the original ISA spec. Rule 12 and the specifics of the atomicity axiom were addressed but not stated in detail.

Other new or modified ISA details:

- There is an RCpc (`.aq` and `.rl`) vs. RCsc (`.aqrl`) distinction
- Load-release and store-acquire are deprecated
- `lr`/`sc` behavior was clarified

A.8.1 Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- 'V' vector ISA extensions
- A transactional memory subset of the 'T' ISA extension
- 'J' JIT extension
- Native encodings for `l{b|h|w|d}.aq`/`s{b|h|w|d}.rl`
- Fences limited to certain addresses
- Cache writeback/flush/invalidate/etc. hints, but these should be considered hints, not functional requirements. Any cache management operations which are required for basic correctness should be described as (possibly address range-limited) fences to comply with the RISC-V philosophy (see also `fence.i` and `sfence.vma`). For example, a functional cache writeback instruction might instead be written as "`fence rw[addr],w[addr]`".

A.9 Litmus Tests

These litmus tests represent some of the better-known litmus tests in the field, plus some tests that are randomly-generated, plus some tests that are generated to be particularly relevant to the RVWMO memory model.

All will be made available for download once they are generated.

We expect that these tests will one day serve as part of a compliance test suite, and we expect that many architects will use them for verification purposes as well.

COMING SOON!

Appendix B

Formal Memory Model Specifications

To facilitate formal analysis of RVWMO, we present a set of formalizations in this chapter. Any discrepancies are unintended; the expectation is that the models will describe exactly the same sets of legal behaviors, pending some memory model changes that have not quite been added to all of the formalizations yet.

As such, these formalizations should be considered snapshots from some point in time during the development process rather than finalized specifications.

At this point, no individual formalization is considered authoritative, but we may designate one as such in collaboration with the ISA specification and/or formalization task groups.

B.1 Formal Axiomatic Specification in Alloy

We present two formal specifications of the RVWMO memory model in Alloy (`http://alloy.mit.edu`).

The first corresponds directly to the natural language model earlier in this chapter.

```
////////////////////////////////////////////////////////////////////////////
// =RISC-V RVWMO axioms=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store

  // explicit synchronization
  + ppo_fence
  + Load.aq <: ^po
  + ^po :> Store.rl
  + Store.aq.rl <: ^po :> Load.aq.rl
  + ^po :> Load.sc
  + Store.sc <: ^po

  // dependencies
  + addr
  + data
  + ctrl :> Store
  + (addr+data).successdep

  // CoRR
  + rdw & po_loc_no_intervening_write

  // pipeline dependency artifacts
  + (addr+data).rfi
  + addr.^po :> Store
  + ctrl.(FenceI <: ^po)
  + addr.^po.(FenceI <: ^po)
}

// the global memory order respects preserved program order
fact { ppo in gmo }
```

Figure B.1: The RVWMO memory model formalized in Alloy (1/4: PPO)

```
// Load value axiom
fun candidates[r: Load] : set Store {
  (r.~gmo & Store & same_addr[r]) // writes preceding r in gmo
  + (r.^~po & Store & same_addr[r]) // writes preceding r in po
}

fun latest_among[s: set Event] : Event { s - s.~gmo }

pred LoadValue {
  all w: Store | all r: Load |
    w->r in rf <=> w = latest_among[candidates[r]]
}

fun after_reserve_of[r: Load] : Event { latest_among[r + r.~rf].gmo }

pred Atomicity {
  all r: Store.~rmw |               // starting from the read r of an atomic,
    no x: Store & same_addr[r + r.rmw] | // there is no write x to the same addr
      x not in same_hart[r]         // from a different hart, such that
      and x in after_reserve_of[r]  // x follows (the write r reads from) in gmo
      and r.rmw in x.gmo            // and r follows x in gmo
}

pred RISCV_mm { LoadValue and Atomicity }
```

Figure B.2: The RVWMO memory model formalized in Alloy (2/4: Axioms)

```
////////////////////////////////////////////////////////////////////////////////
// Basic model of memory

sig Hart {  // hardware thread
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // program order
}

abstract sig MemoryEvent extends Event {
  address: one Address,
  aq: lone MemoryEvent, // opcode bit
  rl: lone MemoryEvent, // opcode bit
  sc: lone MemoryEvent, // for AMOs with .aq and .rl, to distinguish from lr/sc
  gmo: set MemoryEvent   // global memory order
}
sig Load extends MemoryEvent {
  addr: set Event,
  ctrl: set Event,
  data: set Store,
  successdep: set Event,
  rmw: lone Store
}
sig Store extends MemoryEvent {
  rf: set Load
}
sig Fence extends Event {
  pr: lone Fence, // opcode bit
  pw: lone Fence, // opcode bit
  sr: lone Fence, // opcode bit
  sw: lone Fence  // opcode bit
}
sig FenceI extends Event {}

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
    (Load  <: ^po :> FencePRSR).(^po :> Load)
  + (Load  <: ^po :> FencePRSW).(^po :> Store)
  + (Store <: ^po :> FencePWSR).(^po :> Load)
  + (Store <: ^po :> FencePWSW).(^po :> Store)
}
```

Figure B.3: The RVWMO memory model formalized in Alloy (3/4: model of memory)

```
// auxiliary definitions
fun po_loc_no_intervening_write : MemoryEvent->MemoryEvent {
  po_loc - ((po_loc :> Store).po_loc)
}

fun RFInit : Load { Load - Store.rf }
fun rsw : Load->Load { ~rf.rf + (RFInit <: address.~address :> RFInit) }
fun rdw : Load->Load { (Load <: po_loc :> Load) - rsw }

fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// basic facts about well-formed execution candidates
fact { acyclic[po] }
fact { all e: Event | one e.*~po.~start }  // each event is in exactly one hart
fact { rf.~rf in iden } // each read returns the value of only one write
fact { total[gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf in address.~address }
fun rfi : Store->Load { Store <: po_loc_no_intervening_write :> Load }

//dep
fact { addr + ctrl + data in ^po }
fact { successdep in (Write.~rmw) <: ^po }
fact { ctrl.*po in ctrl }
fact { rmw in ^po }

////////////////////////////////////////////////////////////////////////////////
// =Opcode encoding restrictions=

// opcode bits are either set (encoded, e.g., as f.pr in iden) or unset
// (f.pr not in iden).  The bits cannot be used for anything else
fact { pr + pw + sr + sw + aq + rl + sc in iden }
fact { sc in aq + rl }
fact { Load.sc.rmw in Store.sc and Store.sc.~rmw in Load.sc }

// Fences must have either pr or pw set, and either sr or sw set
fact { Fence in Fence.(pr + pw) & Fence.(sr + sw) }

// there is no write-acquire, but there is write-strong-acquire
fact { Store & Acquire in Release }
fact { Load & Release in Acquire }

////////////////////////////////////////////////////////////////////////////////
// =Alloy shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
  all disj e, e': bag | e->e' in rel + ~rel
  acyclic[rel]
}
```

Figure B.4: The RVWMO memory model formalized in Alloy (4/4: Auxiliaries)

The second is an equivalent formulation which is slightly more complex but which is more computationally efficient. We expect that analysis tools will be built off of this second formulation. Also included are empirical checks that the two models match.

This formulation, however, does not apply when mixed-size accesses are used, nor when `lr`/`sc` to different addresses are used.

```
// coherence order: a total order on the writes to each address
fun co : Write->Write { Write <: ((address.~address) & gmo) :> Write }
// from-read: from a read to the coherence successors of the rf-source of the write
fun fr : Read->Write { ~rf.co + ((Read - Write.rf) <: address.~address :> Write) }

// e = external; i.e., from a different hart
fun rfe : Store->Load  { rf - iden - ^po - ^~po }
fun coe : Store->Store { co - iden - ^po - ^~po }
fun fre :  Load->Store { fr - iden - ^po - ^~po }

pred sc_per_location  { acyclic[rf + co + fr + po_loc] }
pred atomicity { no rmw & fre.coe }
pred causality { acyclic[rfe + co + fr + ppo] }

// equality checks
run RISCV_mm_com_sanity { RISCV_mm_com } for 3
check RISCV_mm_gmo_com { RISCV_mm => RISCV_mm_com } for 6
check RISCV_mm_com_gmo {
  rmw in address.~address => // the rf/co/fr model assumes rmw in same addr
  RISCV_mm_com =>
  rfe + co + fr in gmo =>    // pick a gmo which matches rfe+co+fr
  RISCV_mm
} for 6
```

Figure B.5: An alternative, more computationally efficient but less complete axiomatic definition

B.2 Formal Axiomatic Specification in Herd

See also: `http://moscova.inria.fr/~maranget/cats7/riscv`

(This herd model is not yet updated to account for rules 6–7 and 11, and rule `r4` has been tentatively removed from RVWMO. Updates to come...)

```
(*************)
(* Utilities *)
(*************)

let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw


let po-loc-no-w = po-loc \ (po-loc;[W];po-loc)
let rsw = rf^-1;rf

let LD-ACQ = R & (Acq|AcqRel)
and ST-REL = W & (Rel|AcqRel)

(*************)
(* ppo rules *)
(*************)

let r1 = [M];po-loc;[W]
and r2 = fence
and r3 = [LD-ACQ];po;[M]
and r4 = [R];po-loc;[LD-ACQ]
and r5 = [M];po;[ST-REL]
and r6 = [W & AcqRel];po;[R & AcqRel]
and r7 = [R];addr;[M]
and r8 = [R];data;[W]
and r9 = [R];ctrl;[W]
and r10 = ([R];po-loc-no-w;[R]) \ rsw
and r11 = [R];(addr|data);[W];po-loc-no-w;[R]
and r12 = [R];addr;[M];po;[W]
and r13 = [R];ctrl;[Fence.i];po;[R]
and r14 = [R];addr;[M];po;[Fence.i];po;[M]

let ppo =
 r1
| r2
| r3
| r4
| r5
| r6
| r7
| r8
| r9
| r10
| r11
| r12
| r13
| r14
```

Figure B.6: `riscv-defs.cat`, part of a herd version of the RVWMO memory model (1/3)

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(*******************************)
(* Generate global memory order *)
(*******************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write before any write to the same location
  ppo |               # ppo compatible
  rfe                 # first half of

(* Walk over all linear extensions of gmo0 *)
with  gmo from linearisations(M\IW,gmo0)

(* Add initial writes upfront -- convenient for computing rfGMO *)
let gmo = gmo | loc & IW * (M\IW)

(**********)
(* Axioms *)
(**********)

(* Compute rf according to the load value axiom, aka rfGMO *)
let WR = loc & ([W];(gmo|po);[R])
let rfGMO = WR \ (loc&([W];gmo);WR)

(* Check equality of herd rf and of rfGMO *)
empty (rf\rfGMO)|(rfGMO\rf) as RfCons

(* Atomic axion *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext

let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

Figure B.7: `riscv.cat`, a herd version of the RVWMO memory model (2/3)

```
Partial

(***************)
(* Definitions *)
(***************)

(* Define ppo *)
include "riscv-defs.cat"

(* Compute coherence relation *)
include "cos-opt.cat"

(**********)
(* Axioms *)
(**********)

(* Sc per location *)
acyclic co|rf|fr|po-loc as Coherence

(* Main model axiom *)
acyclic co|rfe|fr|ppo as Model

(* Atomicity axiom *)
empty rmw & (fre;coe) as Atomic
```

Figure B.8: `riscv.cat`, part of an alternative herd presentation of the RVWMO memory model (2/3)

Bibliography

[1] RISC-V ELF psABI Specification. `https://github.com/riscv/riscv-elf-psabi-doc/`.

[2] IEEE standard for a 32-bit microprocessor. IEEE Std. 1754-1994, 1994.

[3] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.

[4] Krste Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998. Available as techreport UCB/CSD-98-1014.

[5] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[6] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill Book Company, 1962.

[7] Cray Inc. *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual*, 1.1 edition, June 2003.

[8] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85 –95, 2000.

[9] John M. Frankovich and H. Philip Peterson. A functional description of the Lincoln TX-2 computer. In *Western Joint Computer Conference*, Los Angeles, CA, February 1957.

[10] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[11] J. Goodacre and A.N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42 – 50, 2005.

[12] Linley Gwennap. Digital, MIPS add multimedia extensions. Microprocessor Report, 1996.

[13] Timothy H. Heil and James E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.

[14] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.

[15] Manolis G.H. Katevenis, Robert W. Sherburne, Jr., David A. Patterson, and Carlo H. Séquin. The RISC II micro-architecture. In *Proceedings VLSI 83 Conference*, August 1983.

[16] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 43–54, 2005.

[17] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, Washington, DC, USA, 1998.

[18] David D. Lee, Shing I. Kong, Mark D. Hill, George S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. A VLSI chip set for a multiprocessor workstation–Part I: An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE JSSC*, 24(6):1688–1698, December 1989.

[19] R.B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51 –59, August 1996.

[20] Chris Lomont. Introduction to Intel Advanced Vector Extensions. Intel White Paper, 2011.

[21] Kenichi Miura and Keiichiro Uchida. FACOM Vector Processor System: VP-100/VP-200. In Kawalik, editor, *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, volume F7. Springer-Verlag, 1984. Also in: IEEE Tutorial Supercomputers: Design and Applications. Kai Hwang(editor), pp59-73.

[22] OpenCores. OpenRISC 1000 architecture manual, architecture version 1.0, December 2012.

[23] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443–458, 1981.

[24] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42 –50, August 1996.

[25] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305. IEEE Computer Society, 2001.

[26] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium-III processor. *IEEE Micro*, 20(4):47 –57, 2000.

[27] Balaram Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[28] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, 1965.

[29] M. Tremblay, J.M. O'Connor, V. Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10 –20, August 1996.

[30] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[31] J. Tseng and K. Asanović. Energy-efficient register access. In *Proc. of the 13th Symposium on Integrated Circuits and Systems Design*, pages 377–384, Manaus, Brazil, September 2000.

[32] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *ISCA*, pages 188–197, Ann Arbor, MI, 1984.

[33] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master's thesis, University of California, Berkeley, 2011.

[34] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.

[35] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[36] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.