The RISC-V Instruction Set Manual

Volume I: Unprivileged ISA

Document Version 20180801-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

April 13, 2019

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

Please cite as: "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20180801-draft", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, May 2017.

Preface

This is **a draft of** the next release of the document describing the RISC-V user-level architecture, targeted for release 20180801-draft. 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
RVWMO	0.1	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	0.4	N
N	1.1	N
Zam	0.1	N
Ztso	0.1	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:

- Changed document version scheme to avoid confusion with versions of the ISA modules.
- Changed name of document to refer to "unprivileged" instructions as part of move to separate ISA specifications from platform profile mandates.

- Added clearer and more precise definitions of execution environments and harts.
- Defined instruction-set categories: *standard*, *reserved*, *custom*, *non-standard*, and *non-conforming*.
- Defined the signed-zero behavior of FMIN.*fmt* and FMAX.*fmt*, and changed their behavior on signaling-NaN inputs to conform to the minimumNumber and maximumNumber operations in the proposed IEEE 754-201x specification.
- The memory consistency model, RVWMO, has been defined.
- The "Zam" extension, which permits misaligned AMOs and specifies their semantics, has been defined.
- The "Ztso" extension, which enforces a stricter memory consistency model than RVWMO, has been defined.
- Improvements to the description and commentary.
- Defined the term IALIGN as shorthand to describe the instruction-address alignment constraint.

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 [38], 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 [37] 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 ISA extensions and specialized variants.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- An ISA with support for highly-parallel multicore or manycore implementations, including heterogeneous multiprocessors.
- Optional *variable-length instructions* to both expand available instruction encoding space and to support an optional *dense instruction encoding* for improved performance, static code size, and energy efficiency.
- A fully virtualizable ISA to ease hypervisor development.
- An ISA that simplifies experiments with new privileged architecture designs.

Commentary on our design decisions is formatted as in this paragraph, 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 [25], RISC-II [15], SOAR [34], 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.

The RISC-V manual is structured in two volumes. This volume covers the design of the base *unprivileged* instructions, including optional unprivileged ISA extensions. Unprivileged instructions are those that are generally usable in all privilege modes, though behavior might vary depending on privilege mode. The second volume provides the design of the first ("classic") privileged architecture.

In the unprivileged ISA design, we tried to remove any dependence on particular microarchitectural features or on privileged architecture details. This is both for simplicity and to allow maximum flexibility for alternative microarchitecture or privileged architecture implementations.

1.1 RISC-V Hardware Platform Terminology

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

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

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

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

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

1.2 RISC-V Software Execution Environments and Harts

The behavior of a RISC-V program depends on the execution environment in which it runs. The execution environment defines the initial state of the program, the number and type of harts in the environment, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart, and the handling of any interrupts or exceptions raised during

execution including environment calls. The implementation of a RISC-V execution environment can be pure hardware, pure software, or a combination of hardware and software. For example, opcode traps and software emulation can be used to implement functionality not provided in hardware. Examples of execution environments include:

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

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

The term hart was introduced in the work on Lithe [23, 24] to provide a term to represent an abstract execution resource as opposed to a software thread programming abstraction.

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

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

1.3 RISC-V ISA Overview

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 privileged 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 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 address spaces respectively. Chapter 3 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers. Chapter 5 sketches a future RV128I variant of the base integer instruction set supporting a flat 128-bit 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.

RISC-V has been designed to support extensive customization and specialization. Each base integer ISA can be extended with one or more optional instruction-set extensions, and we divide each RISC-V instruction-set encoding space (and related encoding spaces such as the CSRs) into three disjoint categories: *standard*, *reserved*, and *custom*. Standard encodings are defined by the Foundation, and shall not conflict with other standard extensions for the same base ISA. Reserved encodings are currently not defined but are saved for future standard extensions. We use the term *non-standard* to describe an extension that is not defined by the Foundation. Custom encodings shall never be used for standard extensions and are made available for vendor-specific non-standard extensions. We use the term *non-conforming* to describe a non-standard extension that uses either a standard or a reserved encoding (i.e., custom extensions are *not* non-conforming). Instruction-set extensions may provide slightly different functionality depending on the width of the base integer instruction set. Chapter 24 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 25.

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

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

1.4 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 13 reduces code size by providing compressed 16-bit instructions and relaxes the alignment constraints to allow all instructions (16 bit and 32 bit) to be aligned on any 16-bit boundary to improve code density.

We use the term IALIGN (measured in bits) to refer to the instruction-address alignment constraint the implementation enforces. IALIGN is 32 bits in the base ISA, but some ISA extensions, including the compressed ISA extension, relax IALIGN to 16 bits. IALIGN may not take on any value other than 16 or 32.

We use the term ILEN (measured in bits) to refer to the maximum instruction length supported by an implementation, and which is always a multiple of IALIGN. For implementations supporting only a base instruction set, ILEN is 32 bits. Implementations supporting longer instructions have larger values of ILEN.

Figure 1.1 illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to `11`. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to `00`, `01`, or `10`. 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 are illegal. These instructions are considered to be of minimal length: 16 bits if any 16-bit instruction-set extension is present, otherwise 32 bits. The encoding with bits [ILEN-1:0] all ones is also illegal; this instruction is considered to be ILEN bits long.

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 24, 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.

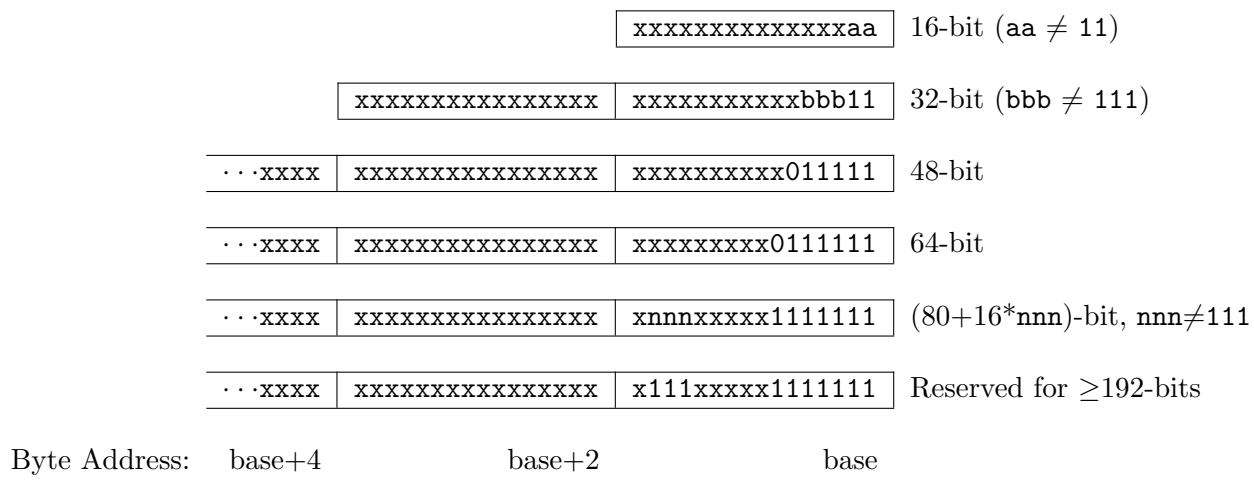


Figure 1.1: RISC-V instruction length encoding.

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.

1.5 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).*

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.

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 [33]. 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).

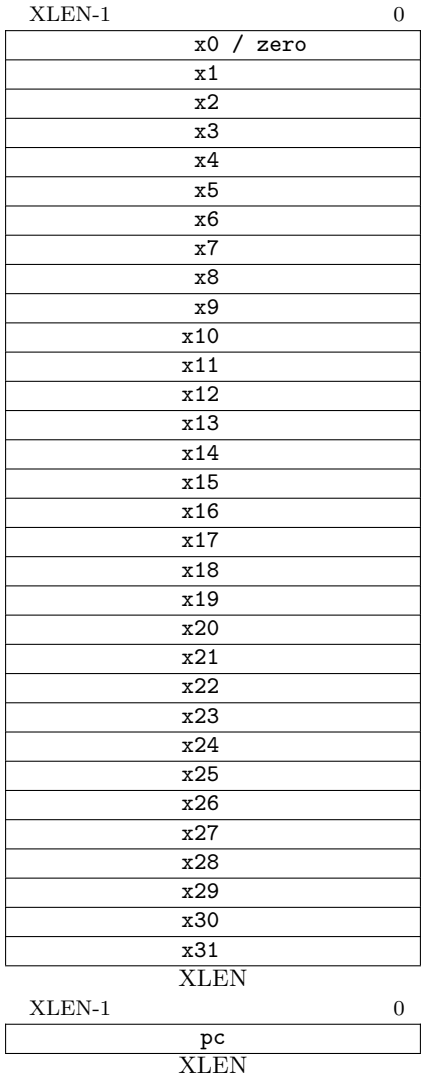


Figure 2.1: RISC-V user-level base integer register state.

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.

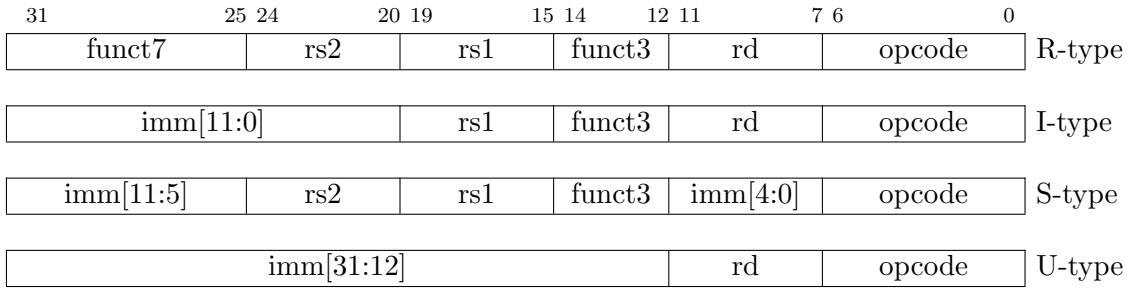


Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

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.

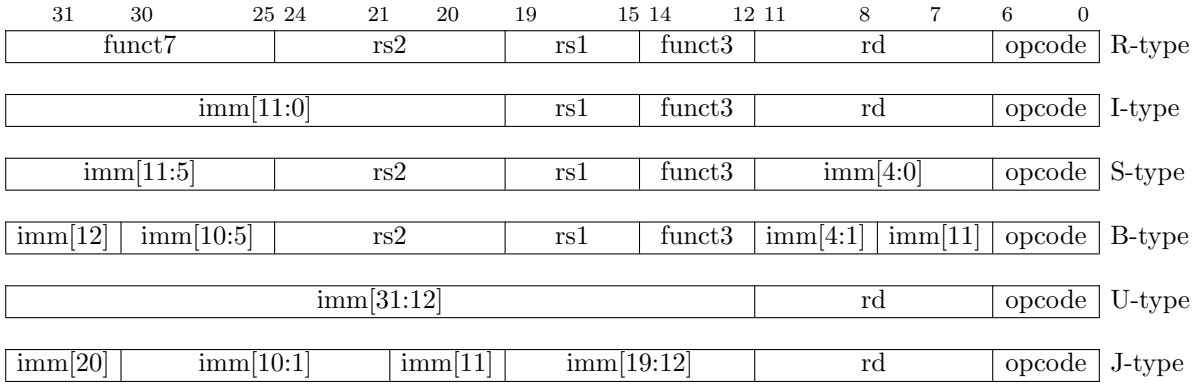


Figure 2.3: RISC-V base instruction formats showing immediate variants.

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.

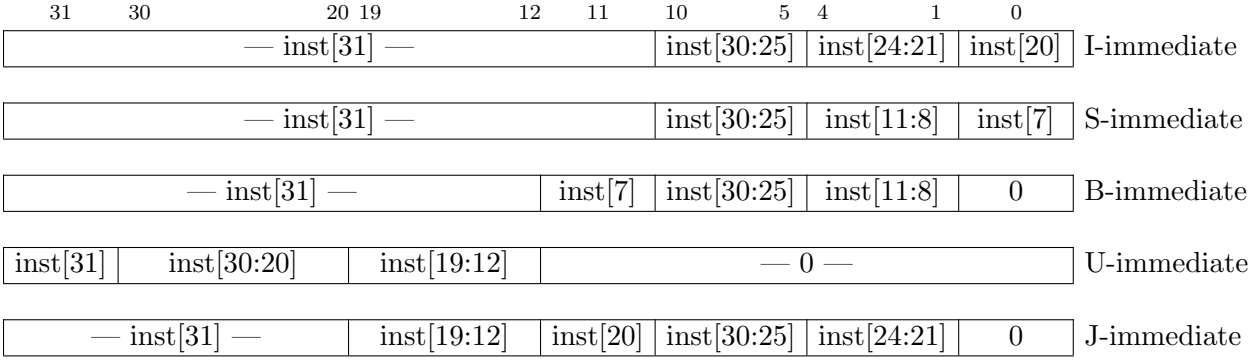


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

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 pseudoinstruction.

SLTI (set less than immediate) places the value 1 in register *rd* if register *rs1* is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to *rd*. SLTIU is

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

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

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

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

NOP Instruction

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

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

NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as for more readable disassembly output. The other NOP encodings are made available for HINT instructions (Section 2.10).

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.

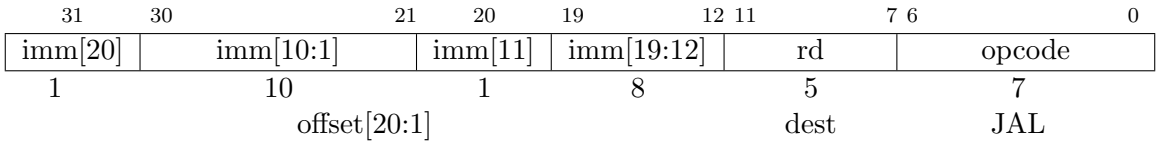
Unconditional Jumps

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

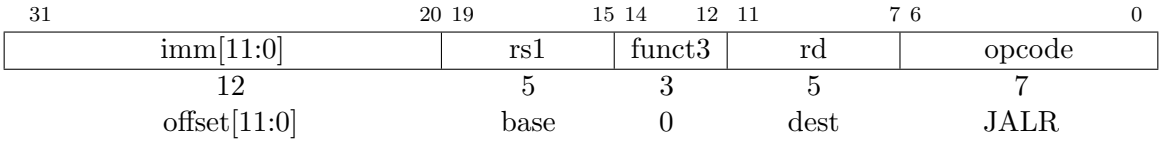
jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (`pc`+4) into register *rd*. The standard software calling convention uses `x1` as the return address register and `x5` as an alternate link register.

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 pseudoinstruction J) are encoded as a JAL with *rd*=`x0`.



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



The unconditional jump instructions all use PC-relative addressing to help support 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, imm12(ra)` *and* `auipc ra, imm20; jalr ra, imm12(ra)`

Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, 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.

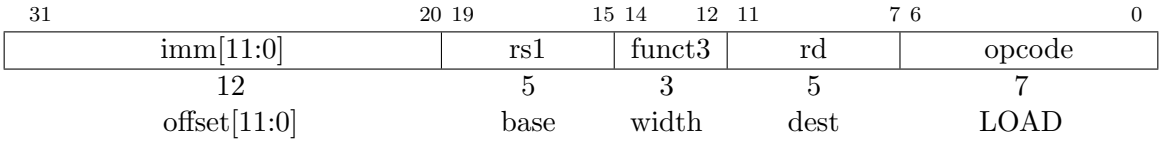
We considered but did not include static branch hints in the instruction encoding. These can reduce the pressure on dynamic predictors, but require more instruction encoding space and software profiling for best results, and can result in poor performance if production runs do not match profiling runs.

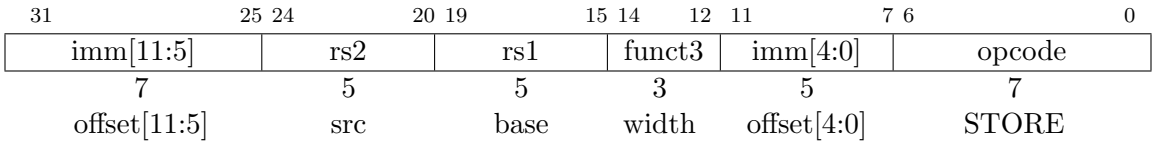
We considered but did not include conditional moves or predicated instructions, which can effectively replace unpredictable short forward branches. Conditional moves are the simpler of the two, but are difficult to use with conditional code that might cause exceptions (memory accesses and floating-point 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 [29]. 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 [29].

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.

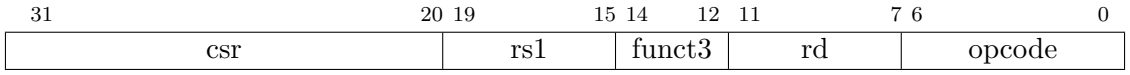
2.7 Control and Status Register Instructions

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.



31 20	19 15	14 12	11 7	6 0
csr	rs1	funct3	rd	opcode
12	5	3	5	7
source/dest	source	CSRRW	dest	SYSTEM
source/dest	source	CSRRS	dest	SYSTEM
source/dest	source	CSRRC	dest	SYSTEM
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=`x0`, then 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 pseudoinstruction to read a CSR, CSRR *rd, csr*, is encoded as CSRRS *rd, csr, x0*. The assembler pseudoinstruction to write a CSR, CSRW *csr, rs1*, is encoded as CSRRW *x0, csr, rs1*, while CSRWI *csr, uimm*, is encoded as CSRRWI *x0, csr, uimm*.

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

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.

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

The RDCYCLE pseudoinstruction reads the low XLEN bits of the `cycle` CSR which holds a count of the number of clock cycles executed by the processor core on which the hart is running from an arbitrary start time in the past. RDCYCLEH is an RV32I-only instruction that reads bits 63–32 of the same cycle counter. The underlying 64-bit counter should never overflow in practice. The rate at which the cycle counter advances will depend on the implementation and operating environment. The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

RDCYCLE is intended to return the number of cycles executed by the processor core, not the hart. Precisely defining what is a "core" is difficult given some implementation choices (e.g., AMD Bulldozer). Precisely defining what is a "clock cycle" is also difficult given the range of implementations (including software emulations), but the intent is that RDCYCLE is used for performance monitoring along with the other performance counters. In particular, where there is one hart/core, one would expect cycle-count/instructions-retired to measure CPI for a hart.

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

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

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

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

The RDTIME pseudoinstruction reads the low XLEN bits of the `time` CSR, which counts wall-clock real time that has passed from an arbitrary start time in the past. RDTIMEH is an RV32I-only instruction that reads bits 63–32 of the same real-time counter. The underlying 64-bit counter should never overflow in practice. The execution environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant. The real-time clocks of all harts in a single user application should be synchronized to within one tick of the real-time clock. The environment should provide a means to determine the accuracy of the clock.

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

The RDINSTRET pseudoinstruction reads the low XLEN bits of the `instret` CSR, which counts the number of instructions retired by this hart from some arbitrary start point in the past. RDINSTRETH is an RV32I-only instruction that reads bits 63–32 of the same instruction counter. The underlying 64-bit counter 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 would like 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

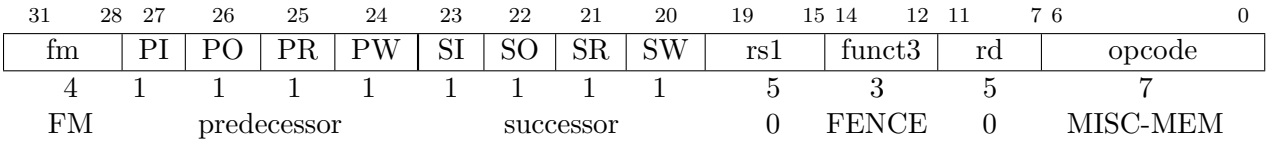
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

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 Ordering Instructions



The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a FENCE before any operation in the *predecessor* set preceding the FENCE. Chapter 6 provides a precise description of the RISC-V memory consistency model.

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.

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

Table 2.2: Fence mode encoding.

The fence mode field *fm* defines the semantics of the FENCE. A FENCE with *fm*=0000 orders all memory operations in its predecessor set before all memory operations in its successor set. A FENCE.TSO instruction orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the FENCE.TSO's predecessor set unordered with non-AMO loads in its successor set.

The unused fields in the FENCE instructions—*rs1* and *rd*—are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields. Likewise, many *fm* and predecessor/successor set settings

in Table 2.2 are also reserved for future use. Base implementations shall treat all such reserved configurations as normal fences with *fm*=0000, and standard software shall use only non-reserved configurations.

We chose a relaxed memory model to allow high performance from simple machine implementations and from likely future coprocessor or accelerator extensions. We separate out I/O ordering from memory R/W ordering to avoid unnecessary serialization within a device-driver hart and also to support alternative non-memory paths to control added coprocessors or I/O devices. Simple implementations may additionally ignore the predecessor *and* successor *fields and always execute a conservative fence on all operations.*

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

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on 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 [32]). 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.

2.10 HINT Instructions

RV32I reserves a large encoding space for HINT instructions, which are usually used to communicate performance hints to the microarchitecture. HINTs are encoded as integer computational

instructions with *rd*=x0. Hence, like the NOP instruction, HINTs do not change any architecturally visible state, except for advancing the pc and any applicable performance counters. Implementations are always allowed to ignore the encoded hints.

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

Table 2.3 lists all RV32I HINT code points. 91% of the HINT space is reserved for standard HINTs, but none are presently defined. The remainder of the HINT space is reserved for custom HINTs: no standard HINTs will ever be defined in this subspace.

No standard hints are presently defined (except the privileged WFI instruction which uses a separately reserved encoding). We anticipate standard hints to eventually include memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation.

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

Table 2.3: RV32I HINT instructions.

Chapter 3

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

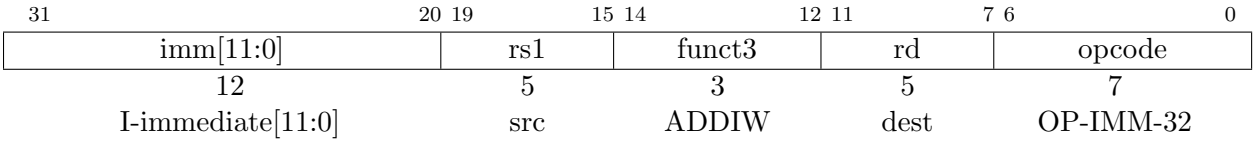
RV64I widens the integer registers and supported user address space to 64 bits (XLEN=64 in Figure 2.1).

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.

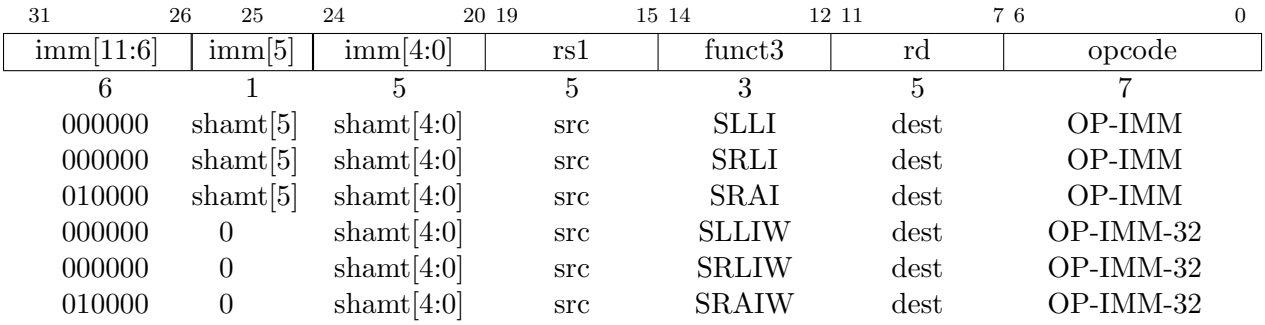
The compiler and calling convention maintain an invariant that all 32-bit values are held in a sign-extended format in 64-bit registers. Even 32-bit unsigned integers extend bit 31 into bits 63 through 32. Consequently, conversion between unsigned and signed 32-bit integers is a no-op, as is conversion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

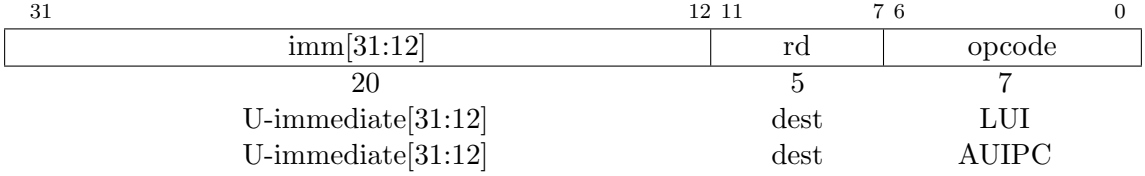
ADDIW is an RV64I-only instruction that adds the sign-extended 12-bit immediate to register *rs1* and produces the proper sign-extension of a 32-bit result in *rd*. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW *rd, rs1, 0* writes the sign-extension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



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

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

4.5 HINT Instructions

All instructions that are microarchitectural HINTs in RV32I (see Section 2.10) are also HINTs in RV64I. The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.

Table 4.1 lists all RV64I HINT code points. 91% of the HINT space is reserved for standard HINTs, but none are presently defined. The remainder of the HINT space is reserved for custom HINTs: no standard HINTs will ever be defined in this subspace.

Instruction	Constraints	Code Points	Purpose
LUI	*rd*=x0	2^{20}	*Reserved for future standard use*
AUIPC	*rd*=x0	2^{20}	
ADDI	*rd*=x0, and either *rs1*≠x0 or *imm*≠0	$2^{17}-1$	
ANDI	*rd*=x0	2^{17}	
ORI	*rd*=x0	2^{17}	
XORI	*rd*=x0	2^{17}	
ADDIW	*rd*=x0	2^{17}	
ADD	*rd*=x0	2^{10}	
SUB	*rd*=x0	2^{10}	
AND	*rd*=x0	2^{10}	
OR	*rd*=x0	2^{10}	
XOR	*rd*=x0	2^{10}	
SLL	*rd*=x0	2^{10}	
SRL	*rd*=x0	2^{10}	
SRA	*rd*=x0	2^{10}	
ADDW	*rd*=x0	2^{10}	
SUBW	*rd*=x0	2^{10}	
SLLW	*rd*=x0	2^{10}	
SRLW	*rd*=x0	2^{10}	
SRAW	*rd*=x0	2^{10}	
SLTI	*rd*=x0	2^{17}	*Reserved for custom use*
SLTIU	*rd*=x0	2^{17}	
SLLI	*rd*=x0	2^{11}	
SRLI	*rd*=x0	2^{11}	
SRAI	*rd*=x0	2^{11}	
SLLIW	*rd*=x0	2^{10}	
SRLIW	*rd*=x0	2^{10}	
SRAIW	*rd*=x0	2^{10}	
SLT	*rd*=x0	2^{10}	
SLTU	*rd*=x0	2^{10}	

Table 4.1: RV64I HINT instructions.

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

RVWMO Memory Consistency Model, Version 0.1

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

Under RVWMO, code running on a single hart appears to execute in order from the perspective of other memory instructions in the same hart, but memory instructions from another hart may observe the memory instructions from the first hart being executed in a different order. Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts. The base RISC-V ISA provides a FENCE instruction for this purpose, described in Section 2.9, while the atomics extension "A" additionally defines load-reserved/store-conditional and atomic read-modify-write instructions.

The standard ISA extension for misaligned atomics "Zam" (Chapter 20) and the standard ISA extension for total store ordering "Ztso" (Chapter 21) augment RVWMO with additional rules specific to those extensions.

The appendices to this specification provide both axiomatic and operational formalizations of the memory consistency model as well as additional explanatory material.

This chapter defines the memory model for regular main memory operations. The interaction of the memory model with I/O memory, instruction fetches, FENCE.I, page table walks, and SFENCE.VMA is not (yet) formalized. Some or all of the above may be formalized in a future revision of this specification. The RV128 base ISA and future ISA extensions such as the "V" vector, "T" transactional memory, and "J" JIT extensions will need to be incorporated into a future revision as well.

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

6.1 Definition of the RVWMO Memory Model

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

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

Memory Model Primitives

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

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

Among instructions in RV32GC and RV64GC, each aligned memory instruction gives rise to exactly one memory operation, with two exceptions. First, an unsuccessful SC instruction does not give rise to any memory operations. Second, FLD and FSD instructions may each give rise to multiple memory operations if XLEN<64, as stated in Section 10.3 and clarified below. An aligned AMO gives rise to a single memory operation that is both a load operation and a store operation simultaneously.

Instructions in the RV128 base instruction set and in future ISA extensions such as V (vector) and P (SIMD) may give rise to multiple memory operations. However, the memory model for these extensions has not yet been formalized.

A misaligned load or store instruction may be decomposed into a set of component memory operations of any granularity. An FLD or FSD instruction for which XLEN<64 may also be decomposed into a set of component memory operations of any granularity. The memory operations generated by such instructions are not ordered with respect to each other in program order, but they are ordered normally with respect to the memory operations generated by preceding and subsequent instructions in program order. The atomics extension "A" does not require execution environments to support misaligned atomic instructions at all; however, if misaligned atomics are supported via the "Zam" extension, LRs, SCs, and AMOs may be decomposed subject to the constraints of the atomicity axiom for misaligned atomics, which is defined in Chapter 20.

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

An LR instruction and an SC instruction are said to be *paired* if the LR precedes the SC in program order and if there are no other LR or SC instructions in between; the corresponding

memory operations are said to be paired as well (except in case of a failed SC, where no store operation is generated). The complete list of conditions determining whether an SC must succeed, may succeed, or must fail is defined in Section 8.2.

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

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

In the memory model literature, the term "RCpc" stands for release consistency with processor-consistent synchronization operations, and the term "RCsc" stands for release consistency with sequentially-consistent synchronization operations [10].

"RCpc" annotations are currently only used when implicitly assigned to every memory access per the standard extension "Ztso" (Chapter 21). Furthermore, although the ISA does not currently contain native load-acquire or store-release instructions, nor RCpc variants thereof, the RVWMO model itself is designed to be forwards-compatible with the potential addition of any or all of the above into the ISA in a future extension.

Syntactic Dependencies

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

In the context of defining dependencies, a "register" refers either to an entire general-purpose register, some portion of a CSR, or an entire CSR. The granularity at which dependencies are tracked through CSRs is specific to each CSR and is defined in Section 6.2.

Syntactic dependencies are defined in terms of instructions' *source registers*, instructions' *destination registers*, and the way instructions *carry a dependency* from their source registers to their destination registers. This section provides a general definition of all of these terms; however, Section 6.3 provides a complete listing of the specifics for each instruction.

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

- In the opcode of i, *rs1*, *rs2*, or *rs3* is set to r
- i is a CSR instruction, and in the opcode of i, *csr* is set to r, unless i is CSRRW or CSRRWI and *rd* is set to `x0`
- r is a CSR and an implicit source register for i, as defined in Section 6.3
- r is a CSR that aliases with another source register for i

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

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

- In the opcode of i, *rd* is set to r
- i is a CSR instruction, and in the opcode of i, *csr* is set to r, unless i is CSRRS or CSRRC and *rs1* is set to `x0` or i is CSRRSI or CSRRCI and uimm[4:0] is set to zero.
- r is a CSR and an implicit destination register for i, as defined in Section 6.3
- r is a CSR that aliases with another destination register for i

Most non-memory instructions *carry a dependency* from each of their source registers to each of their destination registers. However, there are exceptions to this rule; see Section 6.3

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

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

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

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

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

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

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

Preserved Program Order

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

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

- Overlapping-Address Orderings:
  1. b is a store, and a and b access overlapping memory addresses
  2. a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations
  3. a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a
- Explicit Synchronization
  4. There is a FENCE instruction that orders a before b
  5. a has an acquire annotation
  6. b has a release annotation
  7. a and b both have RCsc annotations
  8. a is paired with b
- Syntactic Dependencies
  9. b has a syntactic address dependency on a
  10. b has a syntactic data dependency on a
  11. b is a store, and b has a syntactic control dependency on a
- Pipeline Dependencies
  12. b is a load, and there exists some store m between a and b in program order such that m has an address or data dependency on a, and b returns a value written by m
  13. b is a store, and there exists some instruction m between a and b in program order such that m has an address dependency on a

Memory Model Axioms

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

Load Value Axiom Each byte of each load i returns the value written to that byte by the store that is the latest in global memory order among the following stores:

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

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

The Atomicity Axiom theoretically supports LR/SC pairs of different widths and to mismatched addresses, since implementations are permitted to allow SC operations to succeed in such cases. However, in practice, we expect such patterns to be rare, and their use is discouraged.

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

6.2 CSR Dependency Tracking Granularity

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

Table 6.1: Granularities at which syntactic dependencies are tracked through CSRs

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

6.3 Source and Destination Register Listings

This section provides a concrete listing of the source and destination registers for each instruction. These listings are used in the definition of syntactic dependencies in Section 6.1.

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

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

Key:

[A]Address source register

[D]Data source register

[†]The instruction does not carry a dependency from any source register to any destination register

[‡]The instruction carries dependencies from source register(s) to destination register(s) as specified

RV32I Base Integer Instruction Set

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

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
CSRRW‡	*rs1*, *csr**	*rd*, *csr*		*unless *rd*=x0
CSRRS‡	*rs1*, *csr*	*rd**, *csr*		*unless *rs1*=x0
CSRRC‡	*rs1*, *csr*	*rd**, *csr*		*unless *rs1*=x0

‡carries a dependency from *rs1* to *csr* and from *csr* to *rd*

RV32I Base Integer Instruction Set (continued)

	Source Registers	Destination Registers	Accumulating CSRs	
CSRRWI‡	*csr**	*rd*, *csr*		*unless *rd*=x0
CSRRSI‡	*csr*	*rd*, *csr**		*unless uimm[4:0]=0
CSRRCI‡	*csr*	*rd*, *csr**		*unless uimm[4:0]=0

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

RV64I Base Integer Instruction Set

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

RV32M Standard Extension

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

RV64M Standard Extension

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

RV32A Standard Extension

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

RV64A Standard Extension

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

RV32F Standard Extension

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

RV64F Standard Extension

	Source Registers	Destination Registers	Accumulating CSRs
FCVT.L.S	*rs1*	*rd*	NV, NX
FCVT.LU.S	*rs1*	*rd*	NV, NX
FCVT.S.L	*rs1*	*rd*	NX
FCVT.S.LU	*rs1*	*rd*	NX

RV32D Standard Extension

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

RV64F Standard Extension

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

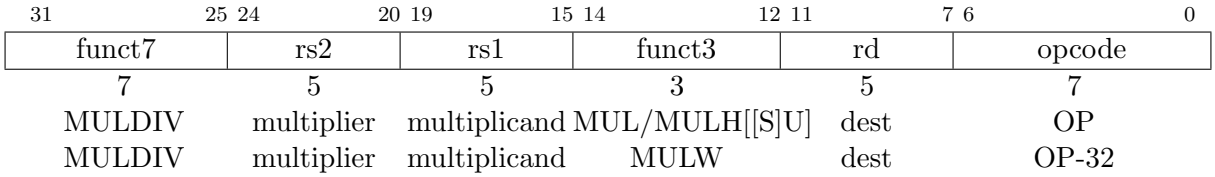
Chapter 7

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

This chapter describes the standard integer multiplication and division instruction extension, which is named "M" and contains instructions that multiply or divide values held in two integer registers.

We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

7.1 Multiplication Operations

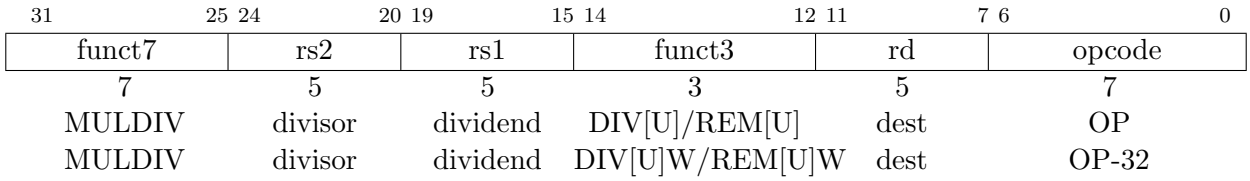


MUL performs an XLEN-bit×XLEN-bit multiplication of *rs1* by *rs2* and places the lower XLEN bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper XLEN bits of the full 2×XLEN-bit product, for signed×signed, unsigned×unsigned, and signed *rs1*×unsigned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

MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the 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.

7.2 Division Operations



DIV and DIVU perform an XLEN bits by XLEN bits signed and unsigned integer division of *rs1* by *rs2*, rounding towards zero. REM and REMU provide the remainder of the corresponding division operation. If both the quotient and remainder are required from the same division, the recommended code sequence is: DIV[U] *rdq, rs1, rs2*; REM[U] *rdr, rs1, rs2* (*rdq* cannot be the same as *rs1* or *rs2*). Microarchitectures can then fuse these into a single divide operation instead of performing two separate divides.

DIVW and DIVUW 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.

The semantics for division by zero and division overflow are summarized in Table 7.1. The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer is divided by −1. The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

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

Table 7.1: Semantics for division by zero and division overflow. L is the width of the operation in bits: XLEN for DIV[U] and REM[U], or 32 for DIV[U]W and REM[U]W.

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.

Chapter 8

"A" Standard Extension for Atomic Instructions, Version 2.0

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.

8.1 Specifying Ordering of Atomic Instructions

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

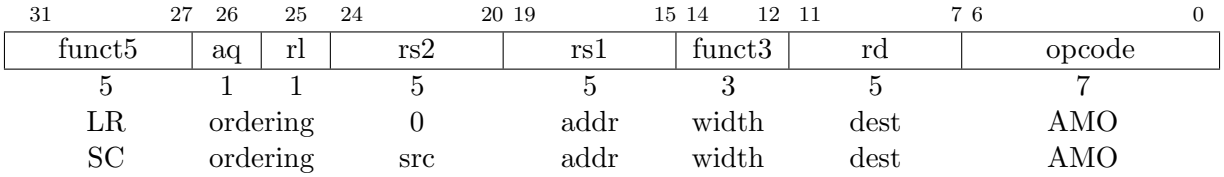
To provide more efficient support for release consistency [10], each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V harts. The bits order accesses to one of the two address domains, memory or I/O, depending on which address 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 cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the *aq* and *rl* bits are set, the atomic memory operation is *sequentially consistent* and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

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.*

8.2 Load-Reserved/Store-Conditional Instructions



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.

For LR and SC, the A extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, a misaligned address exception will be generated.

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.

Although software is not forbidden from using LR/SC sequences that do not meet the forward-progress constraints, portable software must detect the case that the sequence repeatedly fails, then fall back to an alternate code sequence that does not run afoul of the forward-progress constraints. Implementations are permitted to simply always fail any LR/SC sequences that do not meet the forward-progress guarantees.

Certain LR/SC sequences that do not meet the forward progress guarantees may succeed on some implementations. An implementation can reserve an arbitrary subset of the memory space on each LR. An SC may succeed if no store from another hart to the address range reserved by the LR can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. 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 a store from another hart to the address range reserved by the LR can be observed to occur between the LR and the SC. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom in Section 6.1.

In the standard A extension, an SC must fail if there is another SC (to any address) between the LR and the SC in program order. In other words, multiple outstanding reservations are not permitted.

A store-conditional instruction to a scratch word of memory should be used during a preemptive context switch to forcibly yield any existing load reservation.

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 8.1. If inlined, compare-and-swap functionality need only take four 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.
    bnez a0, cas         # Retry if store-conditional failed.
    jr ra                # Return.
fail:
    li a0, 1             # Set return to failure.
    jr ra                # Return.
```

Figure 8.1: Sample code for compare-and-swap function using LR/SC.

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

The *rl* bit on an LR instruction must not be set unless the *aq* bit is also set. The *aq* bit on an SC instruction must not be set unless the *rl* bit is also set.

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".

8.3 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*.

For AMOs, the A extension requires that the address held in *rs1* be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, a misaligned address exception will be generated. The "Zam" extension, described in Chapter 20, relaxes this requirement and specifies the semantics of misaligned AMOs.

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. Setting both the *aq* and the *rl* bit on an AMO makes the sequence sequentially consistent, meaning that it cannot be reordered with earlier or later memory operations from the same hart.

The AMOs were designed to implement the C11 and C++11 memory models efficiently. Although the FENCE R, RW instruction suffices to implement the acquire *operation and FENCE*

RW, W suffices to implement release*, both imply additional unnecessary ordering as compared to AMOs with the corresponding* aq *or* rl *bit set.*

An example code sequence for a critical section guarded by a test-and-set spinlock is shown in Figure 8.2. Note the first AMO is marked *aq* to order the lock acquisition before the critical section, and the second AMO is marked *rl* to order the critical section before the lock relinquishment.

```
    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 8.2: 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 [27].

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 9

"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].

9.1 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 9.1. 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.

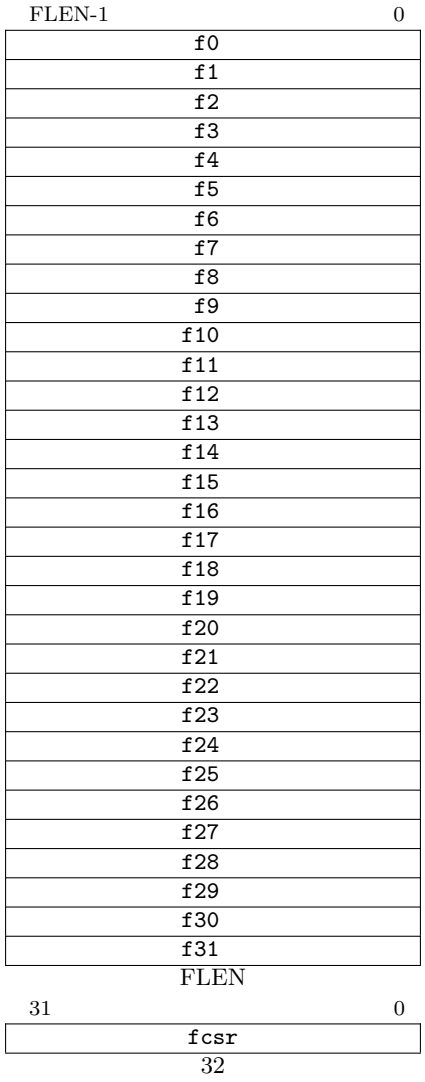


Figure 9.1: RISC-V standard F extension single-precision floating-point state.

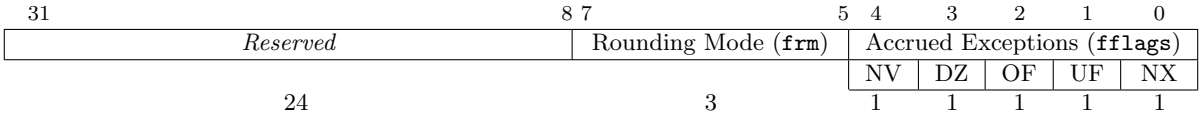


Figure 9.2: Floating-point control and status register.

9.2 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 9.2.

The `fcsr` register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads `fcsr` by copying it into integer register *rd*. FSCSR swaps the value in `fcsr` by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into `fcsr`.

The fields within the `fcsr` can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field `frm` and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in `frm` by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into `frm`. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field `fflags`.

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

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in Table 9.1. 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. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline.

Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
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 9.1: 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 9.2.

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

Table 9.2: 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.

9.3 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.

9.4 Subnormal Arithmetic

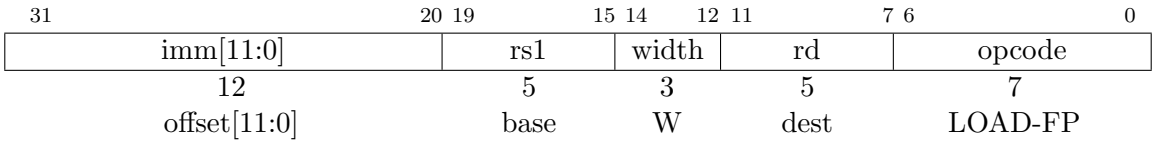
Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

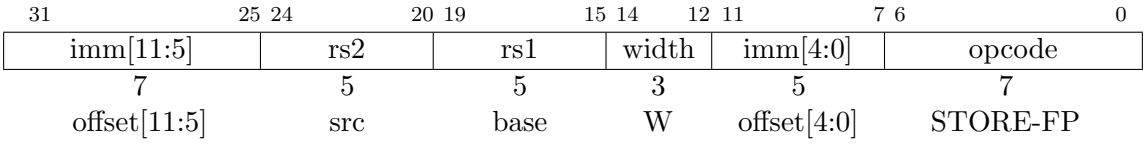
In the parlance of the IEEE standard, tininess is detected after rounding.

Detecting tininess after rounding results in fewer spurious underflow signals.

9.5 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.

9.6 Single-Precision Floating-Point Computational Instructions

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

The 2-bit floating-point format field *fmt* is encoded as shown in Table 9.3. 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 9.1.

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 9.3: Format field encoding.

Floating-point minimum-number and maximum-number instructions FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. For the purposes of these instructions only, the value −0.0 is considered to be less than the value +0.0. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs 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. FMADD.S multiplies the values in *rs1* and *rs2*, adds the value in *rs3*, and writes the final result to *rd*. FMADD.S computes *(rs1×rs2)+rs3*. FMSUB.S multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. FMSUB.S computes *(rs1×rs2)-rs3*. FNMSUB.S multiplies the values in *rs1* and *rs2*, negates the product, adds the value in *rs3*, and writes the final result to *rd*. FNMSUB.S computes *-(rs1×rs2)+rs3*. FNMADD.S multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. FNMADD.S computes *-(rs1×rs2)-rs3*.

The 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 ∞ × 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

9.7 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 9.4 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 9.4: 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 pseudoinstruction FMV.S *rx, ry*); FSGNJN.S *rx, ry, ry* moves the negation of *ry* to *rx* (assembler pseudoinstruction FNEG.S *rx, ry*); and FSGNJX.S *rx, ry, ry* moves the absolute value of *ry* to *rx* (assembler pseudoinstruction FABS.S *rx, ry*).

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

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

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.W moves the single-precision value in floating-point register *rs1* represented in IEEE 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.

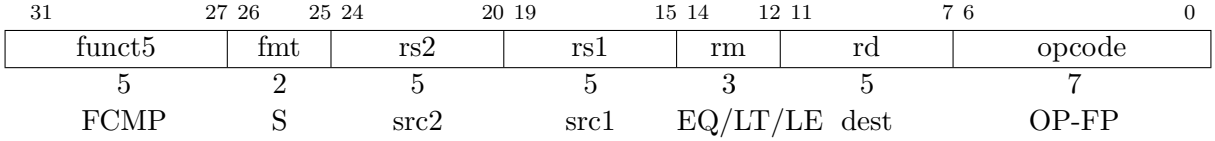
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FMV.X.W	S	0	src	000	dest	OP-FP
FMV.W.X	S	0	src	000	dest	OP-FP

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

9.8 Single-Precision Floating-Point Compare Instructions

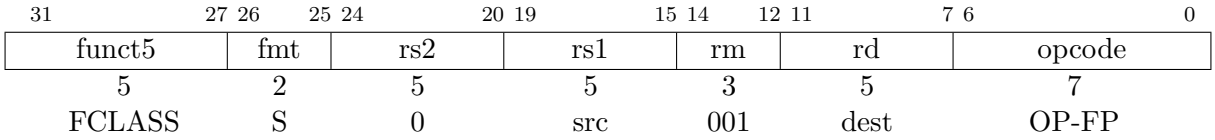
Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers ($rs1 = rs2$, $rs1 < rs2$, $rs1 \leq rs2$) writing 1 to the integer register *rd* if the condition holds, and 0 otherwise.

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.



9.9 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 9.5. 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 9.5: Format of result of FCLASS instruction.

Chapter 10

"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.

10.1 D Register State

The D extension widens the 32 floating-point registers, `f0`–`f31`, to 64 bits (FLEN=64 in Figure 9.1). The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section 10.2.

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.

10.2 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.

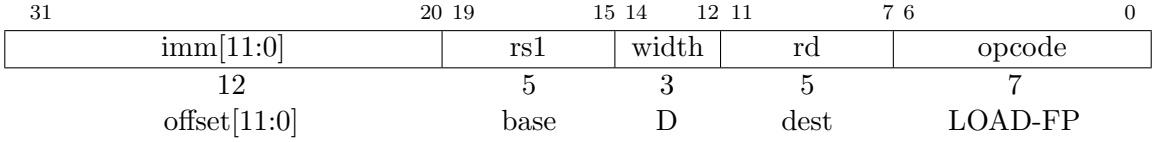
Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in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

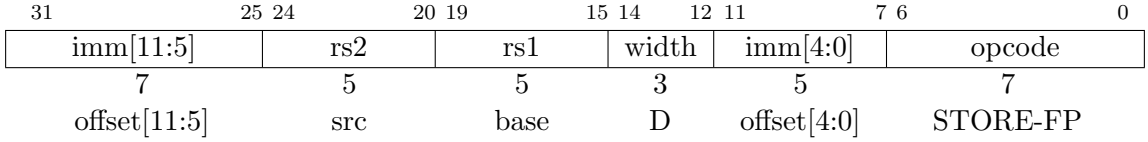
Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can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

10.3 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.

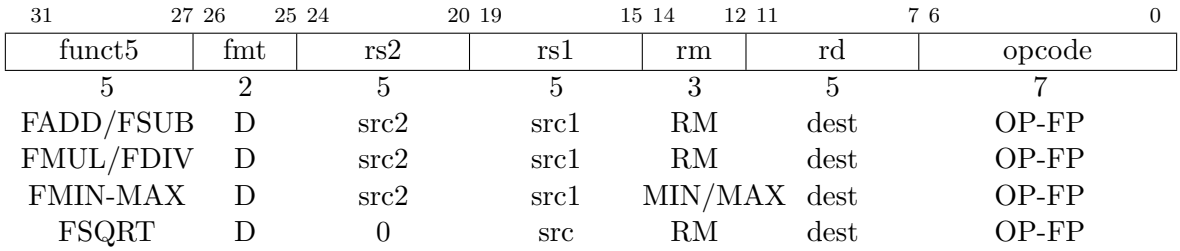


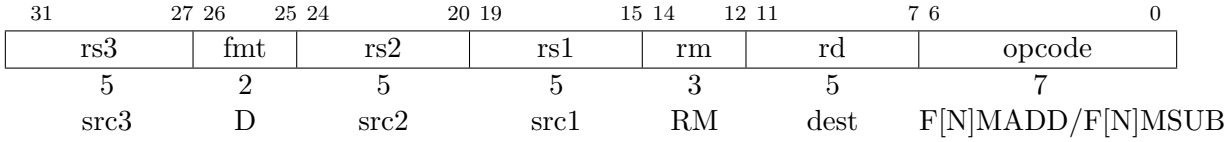


FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64.

10.4 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.





10.5 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.

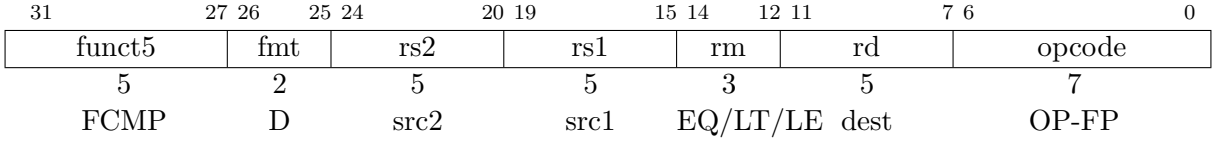
31 27	26 25	24 20	19 15	14 12	11 7	6 0
funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FSGNJ	D	src2	src1	J[N]/JX	dest	OP-FP

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

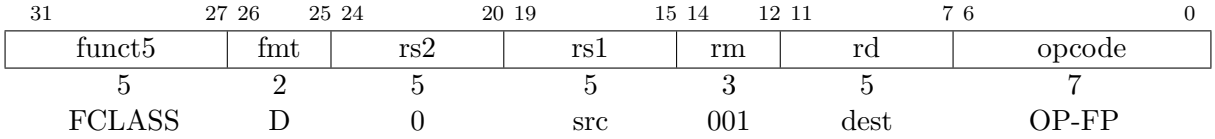
10.6 Double-Precision Floating-Point Compare Instructions

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



10.7 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.



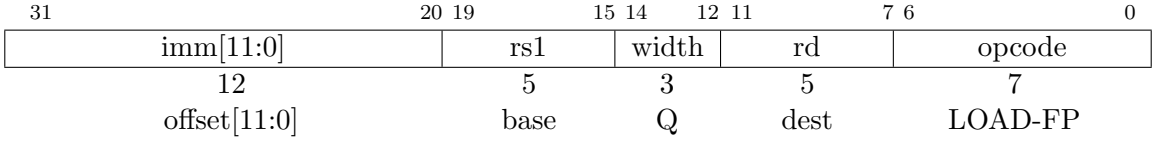
Chapter 11

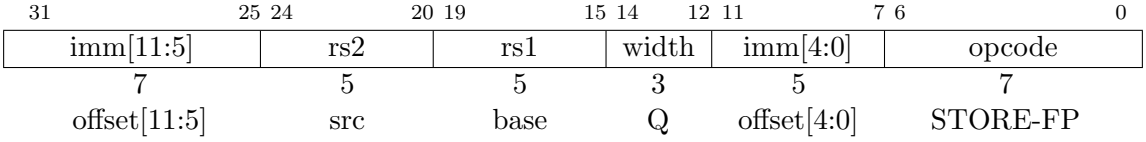
"Q" Standard Extension for Quad-Precision Floating-Point, Version 2.0

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 10.2 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.

11.1 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.

11.2 Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 11.1.

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 11.1: 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

11.3 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.

11.4 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

11.5 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 12

"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.

12.1 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 13

"C" Standard Extension for Compressed Instructions, Version 2.0

This chapter describes the current proposal for the RISC-V standard compressed instruction-set extension, named "C", which reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%–60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%–30% code-size reduction.

13.1 Overview

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions.

Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.

The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 13.4, a few opcodes are used for different purposes depending on base ISA width. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.

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

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

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

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

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

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

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

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

Variable-length instruction sets have long been used to improve code density. For example, the

IBM Stretch [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 [30], 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 [35].*

13.2 Compressed Instruction Formats

Table 13.1 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 13.2 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 13.1: 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 13.2: Registers specified by the three-bit *rs1′*, *rs2′*, and *rd′* fields of the CIW, CL, CS, and CB formats.

13.3 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.LWSP is only valid when *rd*≠`x0`.

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.LDSP is only valid when *rd*≠`x0`.

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.LQSP is only valid when *rd*≠`x0`.

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 [36].

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′)`.

13.4 Control Transfer Instructions

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

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

These instructions use the CJ format.

C.J performs an unconditional control transfer. The offset is sign-extended and added to the `pc` to form the jump target address. C.J can therefore target a ±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]`.

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

These instructions use the CR format.

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

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

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

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

These instructions use the CB format.

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the `pc` to form the branch target address. It can therefore target a ±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]`.

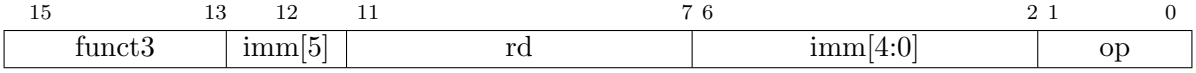
C.BNEZ is defined analogously, but it takes the branch if *rs1′* contains a nonzero value. It expands to `bne rs1′, x0, offset[8:1]`.

13.5 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.



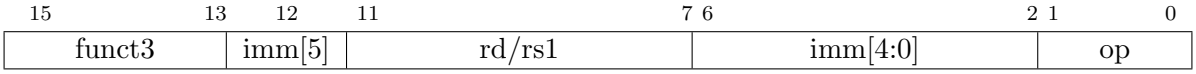
3	1	5	5	2
C.LI	imm[5]	dest≠0	imm[4:0]	C1
C.LUI	nzimm[17]	dest≠{0, 2}	nzimm[16:12]	C1

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.



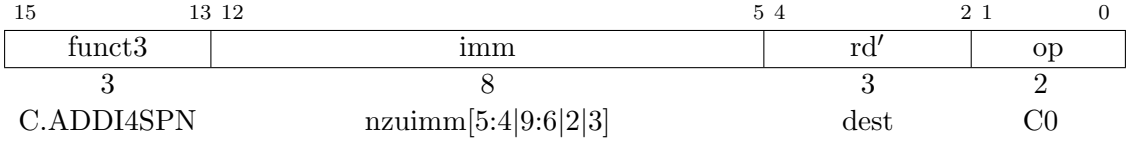
3	1	5	5	2
C.ADDI	nzimm[5]	dest≠0	nzimm[4:0]	C1
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1
C.ADDI16SP	nzimm[9]	2	nzimm[4|6|8:7|5]	C1

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`.

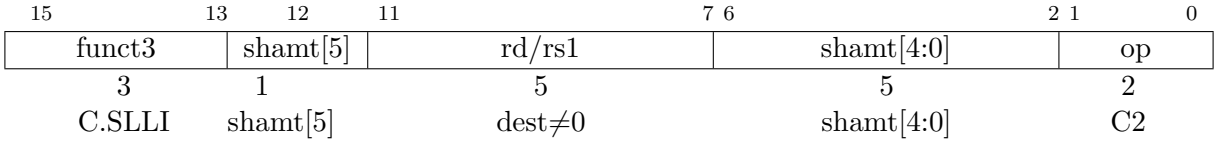
C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into `addiw rd, rd, imm[5:0]`. The immediate can be zero for C.ADDIW, where this corresponds to `sext.w rd`. C.ADDIW is only valid when *rd*≠`x0`.

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]`.

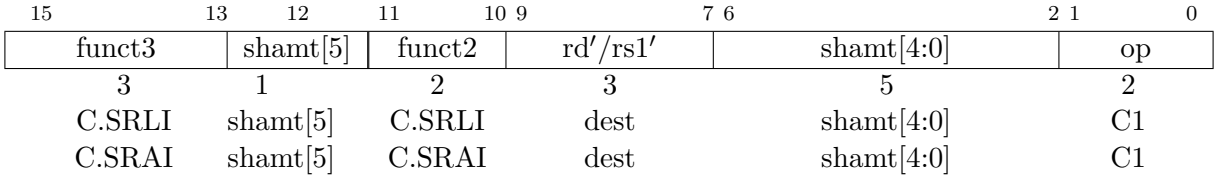
In the standard RISC-V calling convention, the stack pointer `sp` *is always 16-byte aligned.*



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]`.



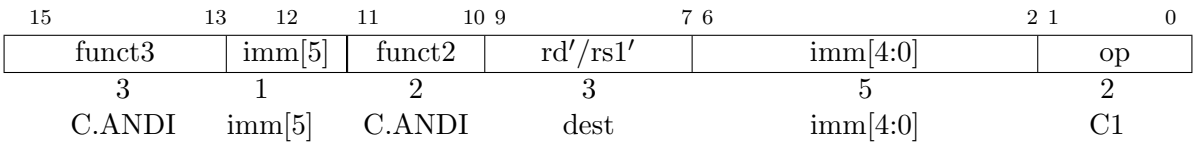
C.SLLI is a CI-format instruction that performs a logical left shift of the value in register *rd* then writes the result to *rd*. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount 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

15 12	11 7	6 2	1 0
funct4	rd/rs1	rs2	op
4	5	5	2
C.MV	dest≠0	src≠0	C2
C.ADD	dest≠0	src≠0	C2

These instructions use the CR format.

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

C.MV 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`.

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

These instructions use the 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′`.

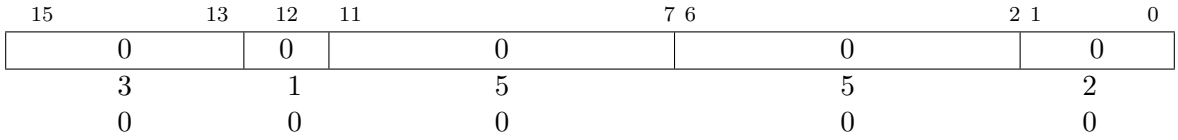
C.SUB subtracts the value in register *rs2′* from the value in register *rd′*, then writes the result to register *rd′*. C.SUB expands into `sub rd′, rd′, rs2′`.

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.

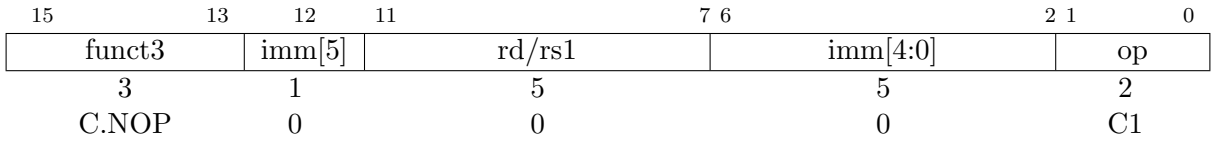
Defined Illegal Instruction



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

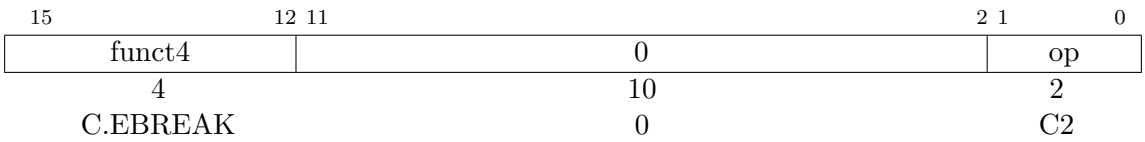
We reserve all-zero instructions to be illegal instructions to help trap attempts to execute zero-ed or non-existent portions of the memory space. The all-zero value should not be redefined in any non-standard extension. Similarly, we reserve instructions with all bits set to 1 (corresponding to very 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



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.

13.6 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 8.2.

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.

13.7 HINT Instructions

A portion of the RVC encoding space is reserved for microarchitectural HINTs. Like the HINTs in the RV32I base ISA (see Section 2.10), these instructions do not modify any architectural state, except for advancing the `pc` and any applicable performance counters. HINTs are executed as no-ops on implementations that ignore them.

RVC HINTs are encoded as computational instructions that do not modify the architectural state, either because *rd*=`x0` (e.g. C.ADD *x0*, *t0*), or because *rd* is overwritten with a copy of itself (e.g. C.ADDI *t0*, 0).

This HINT encoding has been chosen so that simple implementations can ignore HINTs altogether, and instead execute a HINT as a regular computational instruction that happens not to mutate the architectural state.

RVC HINTs do not necessarily expand to their RVI HINT counterparts. For example, C.ADD *x0*, *t0* might not encode the same HINT as ADD *x0*, *x0*, *t0*.

The primary reason to not require an RVC HINT to expand to an RVI HINT is that HINTs are unlikely to be compressible in the same manner as the underlying computational instruction. Also, decoupling the RVC and RVI HINT mappings allows the scarce RVC HINT space to be allocated to the most popular HINTs, and in particular, to HINTs that are amenable to macro-op fusion.

Table 13.3 lists all RVC HINT code points. For RV32C, 78% of the HINT space is reserved for standard HINTs, but none are presently defined. The remainder of the HINT space is reserved for custom HINTs: no standard HINTs will ever be defined in this subspace.

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

Table 13.3: RVC HINT instructions.

13.8 RVC Instruction Set Listings

Table 13.4 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 microarchitectural hints (see Section 13.7).

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

Table 13.4: RVC opcode map

Tables 13.5–13.7 list the RVC instructions.

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

Table 13.5: Instruction listing for RVC, Quadrant 0.

15 14 13	12	11 10	9 8 7	6 5 4 3 2	1 0	
000	nzimm[5]	0		nzimm[4:0]	01	C.NOP *(HINT, nzimm≠0)*
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]	01	C.ADDI *(HINT, nzimm=0)*
001	imm[11|4|9:8|10|6|7|3:1|5]				01	C.JAL *(RV32)*
001	imm[5]	rs1/rd≠0		imm[4:0]	01	C.ADDIW *(RV64/128; RES, rd=0)*
010	imm[5]	rd≠0		imm[4:0]	01	C.LI *(HINT, rd=0)*
011	nzimm[9]	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 13.6: Instruction listing for RVC, Quadrant 1.

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

Table 13.7: Instruction listing for RVC, Quadrant 2.

Chapter 14

"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 15

"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 16

"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 17

"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 [31], MIPS MDMX [12], PowerPC AltiVec [8], Intel x86 MMX/SSE [26, 28], 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 17.1. 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 17.1: 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 18

"V" Standard Extension for Vector Operations, Version 0.4-DRAFT

This version is out-of-date with respect to the current working group draft, which is now hosted on `https://github.com/riscv/riscv-v-spec`.

This chapter presents a proposal for the RISC-V base vector instruction-set extension. 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 base vector extension defines the components that must be included when the "V" bit is set in the `misa` register, and consequently those that will be assumed to exist by software written for an ABI specifying V.

This draft version of the chapter includes additional specifications of proposed extensions to the base vector extension to explain some of the encoding choices made for the base.

The vector extension supports a configurable vector unit, to enable implementations to tradeoff the number of active 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 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.

18.1 Vector Unit State

The additional vector unit architectural state includes 32 vector registers (`v0`–`v31`), and an XLEN-bit WARL vector length CSR, `vl`. Each vector register `v`*n* has an associated 16-bit configuration field `vtype`*n* described below. A 6-bit global maximum element width register `vmaxew` defines the maximum number of bits of storage in every element of every active vector register.

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

Future 2D shape extensions add two more vector length registers, `vm` *and* `vn`*.*

There is also a 3-bit fixed-point rounding mode CSR `vxrm`, and a single-bit fixed-point saturation status CSR `vxsat`. The `vcs` CSR alias provides combined access to the `vl`, `vxrm`, `vxsat` fields to reduce context switch time. The `vcs` register also includes a configuration mode field to support future extended configuration modes.

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

18.2 Vector Unit Type Configuration Register (`vtype`*n*)

The vector unit must be configured before use. Each architectural vector register, `v`*n*, is configured via 16 bits of vector type configuration state `vtype`*n*, which can be accessed via vector configuration (`vcfg`) CSRs and other rapid vector configuration instructions as described below. The vector register type configuration encodes the overall organization, or *shape*, of the elements in each vector register (e.g., scalar versus 1-D vector), as well as the bitwidth and numeric representation of each element. As shown in Figure 18.1, the 16-bit `vtype`*n* encoding is divided into a 5-bit current shape field `vshape`*n*, a 5-bit representation field `verep`*n*, and a 6-bit element bit-width field `vew`*n* held in the `vcfg`*x* CSRs. The combination of an element numeric representation and an element bitwidth is called an element *format*. Each vector register can also be disabled to free physical vector storage for other architectural vector registers.

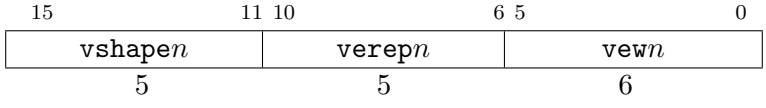


Figure 18.1: Location of subfields within a single `vtype`*n* field.

It was also common in earlier vector machines to support multiple precisions within the vector datapath. In particular, the CDC STAR-100 [?] supported single-precision and double-precision floating-point operations and also bit, byte, and nibble operations in the vector unit; TI ASC [?] designs supported dividing 64-bit vector lanes into two 32-bit lanes for double throughput.

18.3 Shape Encoding

The 5-bit shape field describes the structure of the elements within the vector register. In the base vector extension, the shape can be set to either scalar or vector.

`vshape`	Shape
00000	scalar
00100	1-D vector, length controlled by `vl`
All other encodings reserved	

Table 18.1: Base vector encoding of `vshape`*n* field.

For the base vector ISA, only a single bit is required in each `vshape` *field to select between scalar and 1-D vector elements with the other bits hardwired to zero.*

`vshape`	Shape
00000	scalar
00001	*Reserved*
0001x	*Reserved*
00100	1-D vector `vl`
01000	1-D vector `vm`
01100	1-D vector `vn`
00101	2-D matrix `vl` x `vl`
00110	2-D matrix `vl` x `vm`
00111	2-D matrix `vl` x `vn`
01001	2-D matrix `vm` x `vl`
01010	2-D matrix `vm` x `vm`
01011	2-D matrix `vm` x `vn`
01101	2-D matrix `vn` x `vl`
01110	2-D matrix `vn` x `vm`
01111	2-D matrix `vn` x `vn`
1xxxx	*Reserved/Custom*

Table 18.2: Extended encoding of per-vector-register `vshape` field.

A sketch of the proposed encodings for the 2D shape extension is shown in the Table.

18.4 Representation Encoding

The 5-bit `verep`*n* register sets the numeric representation of each element of the vector register. In the base vector extension, the representation can be set to unsigned integer, two's-complement signed integer, or floating-point. The floating-point representations follow the IEEE 754 standards.

`verep`	Representation
00000	Unsigned integer
00001	Two's-complement signed integer
00010	*Reserved (unsigned floating-point?)*
00011	IEEE-754 floating-point
All other encodings reserved	

Table 18.3: Base vector representation encoding.

`verep`	Representation
00000	Unsigned integer
00001	Two's-complement signed integer
00010	*Reserved (unsigned floating-point)*
00011	IEEE-754 floating-point
001x0	*Reserved*
00101	Complex signed integer
00111	Complex floating-point
01000	Prime Galois field - integer representation
01001	Prime Galois field - Montgomery representation
01100	Binary extension Galois field - polynomial basis
01101	Binary extension Galois field - normal basis
01010	UNORM
01011	SNORM
01110	*Reserved*
01111	*Reserved (complex SNORM?)*
10xxx	Custom representations
11xxx	*Reserved*

Table 18.4: Extended vector representation encoding.

The complex representations split the element width given in `vew`*n into two equal-sized real and imaginary fields, so an element width of 64 bits can hold a single complex value with a 32-bit real and a 32-bit imaginary component.*

18.5 Element Bitwidth

Each vector register, v*n*, has a 6-bit element width register, vew*n*, to specify the number of bits for each element of the current type in the vector register.

The largest element width supported is termed ELEN, and is defined to be the larger of the supported integer and floating-point type widths:

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

For the base vector ISA, the bit width can be set at any power of two between 8 and ELEN.

vew	Width	Required in Base
000 000	disabled	All
001 000	8	All
010 000	16	All
011 000	32	All
100 000	64	RV32D, RV64, RV128
101 000	128	RV64Q, RV128
All other encodings reserved.		

Table 18.5: Base vector ISA encoding of vector element width (vew*n*) register fields.

The extended bit-width encoding is designed to minimize the number of state bits required to support useful subsets of widths. For example, an RV32 system only needs two bits of state per vew*n field to represent* disabled*, 8, 16, and 32. An RV32 system with 3 bits of state can represent* disabled*, 4, 8, 12, 16, 24, 32, and 48. An RV64 system with 4 bits of state can represent* disabled*, 4, 8, 12, 16, 24, 32, 48, 64, 96, 128, 256, 512, 1024.*

vew	Width
000 000	disabled
000 001	1
000 xxx	steps of 1
000 111	7
001 000	8
001 xxx	steps of 1
001 111	15
010 000	16
010 xxx	steps of 2
010 111	30
011 000	32
011 xxx	steps of 4
011 111	60
100 000	64
100 xxx	steps of 8
100 111	120
101 xxx	reserved
110 000	128
110 001	192
110 010	2048
110 011	3072
110 100	512
110 101	768
110 110	8192
110 111	12288
111 000	256
111 001	384
111 010	4096
111 011	6144
111 100	1024
111 101	1536
111 110	16384
111 111	24576

Table 18.6: Proposed extended encoding of vector element width (`vew`n) register fields. Every bit width between 1 and 16 can be supported. Bit widths in steps of 2 between 16 to 32 (i.e., 16, 18, 20, ...). Bit widths in steps of 4 between 32 to 64 (i.e., 32, 36, 40, ...). Bit widths in steps of 8 between 64 and 128 (i.e., 64, 72, 80,...). For bit widths greater than 128, all powers-of-two up to 16384 and all widths 1.5× greater are supported (128, 384, 512, 768,...).

18.6 Base Vector Extension Supported Types

The types supported by the base V extension depend upon the base scalar ISA and supported extensions. When the base V extension is added to a base scalar ISA, it must support the vector data element types implied by the supported scalar types as defined by Table 18.7.

Supported Fixed-Point Formats	
RV32I	I8, U8, I16, U16, I32, U32
RV64I	I8, U8, I16, U16, I32, U32, I64, U64
RV128I	I8, U8, I16, U16, I32, U32, I64, U64, I128, U128

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

Table 18.7: Supported data element formats depending on base integer ISA and supported floating-point extensions. Ix indicates a signed integer of x bits, Ux indicates an unsigned integer of x bits, and Fx indicates an IEEE floating-point number of x bits.

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

18.7 Maximum Vector Element Width (vmaxew)

The global vmaxew field is used to support more complex vector runtime environments where the types to be held in each register of a single configuration may vary dynamically, and may not even be known at compile time due to separate compilation.

The global maximum element width register vmaxew defines the maximum number of bits of storage in every element of every active architectural register, or if zero, defers to the per-vector-register width field.

The VIRAM processor had a virtual processor width register similar to vmaxew *[**?**].*

If vmaxew is zero, then the per-element vector element widths vew*n* determine the minimum storage required for each element of the associated vector register v*n*.

If vmaxew is non-zero, it sets the largest element width that can be supported in any vector register element in the current configuration.

18.8 Vector Configuration Registers (vcfg0–vcfg15)

The vector type configuration requires 512 bits of state (32 vector registers each with 16-bit vtype$n$ field) that can be accessed via the vcfg CSRs.

RV128 uses four vector configuration CSRs: vcfg0 holds configuration data for v0–v7 with bits $16n$ to $16n+15$ holding vtype$n$, while vcfg4, vcfg8 and vcfg12 similarly holds configuration data for v8–v15, v16–v23, and v24–v31 respectively.

In RV64, the vcfg2 CSR provides access to the upper 64 bits of vcfg0 and vcfg6 provides access to the upper 64 bits of vcfg4. In RV32, the vcfg1, vcfg3, vcfg5 and vcfg7 CSRs provides access to the upper bits of vcfg0, vcfg2, vcfg4 and vcfg6 respectively.

Any CSR write to a vcfg$x$ register zeros all vcfg$y$ registers, for $y > x$. As a result configuration data should be written from the vcfg0 CSR upwards.

Zeroing higher-numbered vcfg$y$ *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; an entire old configuration must be read out before a new configuration is written in.*

Additional instructions are provided to support more rapid changes to the vector unit configuration as described below.

18.9 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. An exception will be raised if an instruction tries to change vtype$n$ 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 is also simplified with this constraint.

18.10 Vector Unit CSRs

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
vmaxew	TBD	RV32, RV64, RV128	Global maximum vector element width
vcfg0	TBD	RV32, RV64, RV128	Vector register configuration
vcfg1	TBD	RV32	
vcfg2	TBD	RV32, RV64	
vcfg3	TBD	RV32	
vcfg4	TBD	RV32, RV64, RV128	
vcfg5	TBD	RV32	
vcfg6	TBD	RV32, RV64	
vcfg7	TBD	RV32	
vcfg8	TBD	RV32, RV64, RV128	
vcfg9	TBD	RV32	
vcfg10	TBD	RV32, RV64	
vcfg11	TBD	RV32	
vcfg12	TBD	RV32, RV64, RV128	
vcfg13	TBD	RV32	
vcfg14	TBD	RV32, RV64	
vcfg15	TBD	RV32	

Table 18.8: Vector extension CSRs.

18.11 Maximum Vector Length (MVL)

The implementation determines an available *maximum vector length* (MVL) dependent on the current vector type configuration held in `vcfg`*x* and `vmaxew`. The available MVL depends on the configuration setting and on the implementation's microarchitecture, but MVL must always have the same value for the same configuration parameters on a given hart.

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] supported combining power-of-2 base vector registers into longer vector registers.

The Scale [], Maven [], and Hwacha [] processors also support configuration-dependent MVL.

Previously, the specification imposed a minimum vector length (4) on all configurations to allow stripmining code to be removed for short vector lengths. With the expanded scope of the vector unit types, this would be too onerous to support, and so the requirement is removed.

Discussion: *A separate mechanism for supporting fixed vector lengths should be designed, possibly as part of an optional extension.*

Any change to the vector configuration that might change MVL cause the entire vector unit state to be zeroed. Any write to the global `vmaxew` causes the entire vector unit state to be zeroed, even if the value in `vmaxew` is unchanged.

If `vmaxew` is non-zero, any write to an individual `vew`*n* register that would set the width greater than `vmaxew` raises an illegal instruction exception and leaves the vector unit state unchanged.

If `vmaxew` is non-zero, any write to an individual `vew`*n* field with a value less than or equal to the value in `vmaxew` only zeros the associated vector register `v`*n* and leaves other vector unit state unchanged. The vector register data is zeroed even if `vew`*n* would be unchanged by the write.

If `vmaxew` is zero, then any write to an individual `vew`*n* register zeros the associated `v`*n* vector register. In addition, any write that changes the value in `vew`*n*, zeros the entire vector unit state.

The state is zeroed to hide implementation-dependent bit mappings and to provide additional security when context swapping. Zero is also a convenient initial value for some loops.

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.

Each vector register can be reconfigured dynamically to hold different formats without zeroing the entire vector unit state provided that: if `vmaxew` is zero, the bit-width of the new format is the same as the current `vew`; or if `vmaxew` is non-zero, the format does not require more than `vmaxew` bits. Any change to a vector register's format zeros the affected vector register.

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

Vector registers have both a maximum element width and a current element data type to allow the same vector 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 `vmaxew` 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* `vmaxew`*n width, even if they internally implement a larger physical* `vmaxew`*n that could accommodate the* `vetype`*n request.*

Discussion: *We can either have 1) implementations raise exceptions whenever illegal values are written to* `vmaxew` *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.*

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

The simplest is max-width-per-implementation *(MWPI), where the vector unit is organized in fixed ELEN-width physical lanes, and changes to* `vmaxew` *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* `vmaxew` *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* `vmaxew` *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* `vmaxew` *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.*

Following Sections are out-of-date.

18.12 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 18.2. 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.

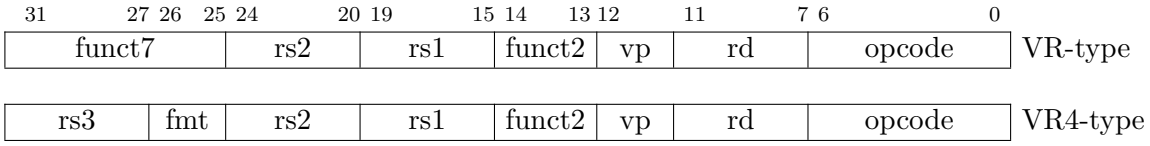


Figure 18.2: New V extension instruction formats.

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.*

18.13 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.

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 18.9: 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.

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 18.9 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.

18.14 Rapid Configuration Instructions

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

The `vsetdcfg` instruction takes a scalar register value encoded as shown in Figure 18.3, 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 `vmaxew`n values set accordingly as shown in Table **??**. 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.

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

Figure 18.3: 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 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.

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.*

18.15 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.

18.16 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 (`vcfg`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

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

Table 18.10: Operation of `setvl` instruction to set vector length register `vl` based on requested application vector length (AVL) and current maximum vector length (MVL).

`setvl` instruction calculates the value to assign to `vl` according to Table 18.10. 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.*

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.*

```
        # 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 18.4: Example vector-vector add loop.

18.17 Predicated Execution

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 explicitly 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.

18.18 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.

18.19 Vector Register Gather

A vector register gather 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
vrgather  vdest, vsrc, vindices
```

18.20 Vector Slide

Reductions (and convolutions) are supported via a vector slide 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.

18.21 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.

18.22 Optional Transcendental Support

Chapter 19

"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.

19.1 Additional CSRs

The user-visible CSRs added to support the N extension are listed in Table 19.1.

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 19.1: CSRs for N extension.

19.2 User Status Register (ustatus)

The ustatus register is an XLEN-bit read/write register formatted as shown in Figure 19.1. The ustatus register keeps track of and controls the hart's current operating state.

Figure 19.1: 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.

19.3 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.

19.4 N Extension Instructions

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

19.5 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 20

"Zam" Standard Extension for Misaligned Atomics, v0.1

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

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

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

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

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

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

Chapter 21

"Ztso" Standard Extension for Total Store Ordering, v0.1

This chapter defines the "Ztso" extension for the RISC-V Total Store Ordering (RVTSO) memory consistency model. RVTSO is defined as a delta from RVWMO, which is defined in Chapter 6.1.

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

RVTSO makes the following adjustments to RVWMO:

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

These rules render all PPO rules except 4–7 redundant. They also make redundant any non-I/O fences that do not have both PW and SR set. Finally, they also imply that no instruction will be reordered past an AMO in either direction.

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

Chapter 22

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

Table 22.1 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 13 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 24 has a more extensive discussion of ways to add extensions to the RISC-V ISA.

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

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	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

31 28	27	26 25	24 20	19 15	14	13 12	11 7	6 0	
RV32V Standard Extension (*draft proposal, subject to change*)									
1000	000		vs2	vs1	1	m	vd	1100111	VADD
1000	001		vs2	vs1	1	m	vd	1100111	VSUB
1001	000		vs2	vs1	1	m	vd	1100111	VSL
1101	000		vs2	vs1	1	m	vd	1100111	VSR
1111	000		vs2	vs1	1	m	vd	1100111	VAND
1110	000		vs2	vs1	1	m	vd	1100111	VOR
1100	000		vs2	vs1	1	m	vd	1100111	VXOR
1001	100		vs2	vs1	1	m	vd	1100111	VSEQ
1001	101		vs2	vs1	1	m	vd	1100111	VSNE
1001	110		vs2	vs1	1	m	vd	1100111	VSLT
1001	111		vs2	vs1	1	m	vd	1100111	VSGE
1011	000		rs2	vs1	1	m	vd	1100111	VCLIP
1011	001		rs2	vs1	1	m	vd	1100111	VCVT
1010	111		00001	vs1	1	m	rd	1100111	VMPOP
1010	111		00000	vs1	1	m	rd	1100111	VMFIRST
1010	000		rs2	vs1	1	m	rd	1100111	VEXTRACT
1011	100		rs2	rs1	1	m	vd	1100111	VINSERT
1100	001		vs2	vs1	1	m	vd	1100111	VMERGE
1100	010		vs2	vs1	1	m	vd	1100111	VSELECT
1011	010		rs2	vs1	1	m	vd	1100111	VSLIDE
1000	100		vs2	vs1	1	m	vd	1100111	VDIV
1000	101		vs2	vs1	1	m	vd	1100111	VREM
1000	110		vs2	vs1	1	m	vd	1100111	VMUL
1000	111		vs2	vs1	1	m	vd	1100111	VMULH
1000	010		vs2	vs1	1	m	vd	1100111	VMIN
1000	011		vs2	vs1	1	m	vd	1100111	VMAX
1001	010		vs2	vs1	1	m	vd	1100111	VSGNJ
1001	011		vs2	vs1	1	m	vd	1100111	VSGNJN
1001	001		vs2	vs1	1	m	vd	1100111	VSGNJX
1100	111		00010	vs1	1	m	vd	1100111	VSQRT
1100	111		00000	vs1	1	m	vd	1100111	VCLASS
1100	111		00001	vs1	1	m	vd	1100111	VPOPC
0000	imm[7:0]			vs1	1	m	vd	1100111	VADDI
0001	imm[7:0]			vs1	1	m	vd	1100111	VSLI
0101	imm[7:0]			vs1	1	m	vd	1100111	VSRI
0111	imm[7:0]			vs1	1	m	vd	1100111	VANDI
0110	imm[7:0]			vs1	1	m	vd	1100111	VORI
0100	imm[7:0]			vs1	1	m	vd	1100111	VXORI
0011	imm[7:0]			vs1	1	m	vd	1100111	VCLIPI
vs3		00	vs2	vs1	0	m	vd	1100111	VMADD
vs3		01	vs2	vs1	0	m	vd	1100111	VMSUB
vs3		11	vs2	vs1	0	m	vd	1100111	VNMADD
vs3		10	vs2	vs1	0	m	vd	1100111	VNMSUB

31 28	27	26 25	24 20	19 15	14	13 12	11 7	6 0	
RV32V Standard Extension (cont.)									
imm[4:0]	00		00000	rs1	1	m	vd	0000111	VLD
imm[4:0]	01		rs2	rs1	1	m	vd	0000111	VLDS
imm[4:0]	10		vs2	rs1	1	m	vd	0000111	VLDX
vs3	00		00000	rs1	1	m	imm[4:0]	0100111	VST
vs3	01		rs2	rs1	1	m	imm[4:0]	0100111	VSTS
vs3	10		vs2	rs1	1	m	imm[4:0]	0100111	VSTX
vs3	11		vs2	00001	1	m	vd	0100111	VAMOSWAP
vs3	11		vs2	00000	1	m	vd	0100111	VAMOADD
vs3	11		vs2	01100	1	m	vd	0100111	VAMOAND
vs3	11		vs2	01000	1	m	vd	0100111	VAMOOR
vs3	11		vs2	00100	1	m	vd	0100111	VAMOXOR
vs3	11		vs2	10000	1	m	vd	0100111	VAMOMIN
vs3	11		vs2	10100	1	m	vd	0100111	VAMOMAX

Table 22.2: Instruction listing for RISC-V

Table 22.3 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 22.3: RISC-V control and status register (CSR) address map.

Chapter 23

RISC-V Assembly Programmer's Handbook

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

Table 23.1 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 23.1: Assembler mnemonics for RISC-V integer and floating-point registers.

Tables 23.2 and 23.3 contain a listing of standard RISC-V pseudoinstructions.

pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, delta[31:12] + delta[11]` `addi rd, rd, delta[11:0]`	Load absolute address, where `delta = symbol − pc`
`l{b|h|w|d} rd, symbol`	`auipc rd, delta[31:12] + delta[11]` `l{b|h|w|d} rd, delta[11:0](rd)`	Load global
`s{b|h|w|d rd, symbol, rt`	`auipc rt, delta[31:12] + delta[11]` `s{b|h|w|d} rd, delta[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, delta[31:12] + delta[11]` `fl{w|d} rd, delta[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, delta[31:12] + delta[11]` `fs{w|d} rd, delta[11:0](rt)`	Floating-point store global

The base instructions use `pc`*-relative addressing, so the linker subtracts* `pc` *from* `symbol` *to get* `delta`*. The linker adds* `delta[11]` *to the 20-bit high part, counteracting sign extension of the 12-bit low part.*

`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 ≠ 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 ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ 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 ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`call offset`	`auipc x1, offset[31:12] + offset[11]` `jalr x1, offset[11:0](x1)`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12] + offset[11]` `jalr x0, offset[11:0](x6)`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Table 23.2: 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 23.3: Pseudoinstructions for accessing control and status registers.

Chapter 24

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.

24.1 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 24.1 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 24.1: 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 24.2: Two-dimensional characterization of standard instruction-set extensions.

Table 24.2 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.

24.2 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.

24.3 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 22.1), 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 22.1).

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.

24.4 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.

24.5 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 25

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.

25.1 Case Sensitivity

The ISA naming strings are case insensitive.

25.2 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.

25.3 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.

25.4 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".

25.5 Underscores

Underscores "_" may be used to separate ISA subset components to improve readability and to provide disambiguation. For example, "RV32I2_M2_A2_F2_D2".

25.6 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 an underscore. For example, an ISA with non-standard extensions Argle and Bargle may be named "RV64GXargle_Xbargle".

25.7 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 an underscore.

25.8 Supervisor-level Extensions

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

25.9 Subset Naming Convention

Table 25.1 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 25.1: 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 26

History and Acknowledgments

26.1 "Why Develop a new ISA?" Rationale from Berkeley Group

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

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

- **Commercial ISAs are proprietary.** Except for SPARC V8, which is an open IEEE standard [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.

26.2 History from Revision 1.0 of ISA manual

The RISC-V ISA and instruction-set manual builds upon several earlier projects. Several aspects of the supervisor-level machine and the overall format of the manual date back to the T0 (Torrent-0) vector microprocessor project at UC Berkeley and ICSI, begun in 1992. T0 was a vector processor based on the MIPS-II ISA, with Krste 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.

26.3 History from Revision 2.0 of ISA manual

Multiple implementations of RISC-V processors have been completed, including several silicon fabrications, as shown in Figure 26.1.

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 26.1: 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 [35].

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.

26.4 History from Revision 2.1

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

Acknowledgments

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

26.5 History from Revision 2.2

Acknowledgments

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

26.6 History for Revision 2.3

Uptake of RISC-V continues at breakneck pace.

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

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

26.7 Funding

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

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

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

Appendix A

RVWMO Explanatory Material, Version 0.1

This section provides more explanation for RVWMO (Chapter 6), using more informal language and concrete examples. These are intended to clarify the meaning and intent of the axioms and preserved program order rules. This appendix should be treated as commentary; all normative material is provided in Chapter 6 and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in Section A.7. Any other discrepancies are unintentional.

A.1 Why RVWMO?

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

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

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

(e.g., to facilitate porting from x86) or whether to instead prioritize compatibility with other RISC-V cores implementing RVWMO.

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

A.2 Litmus Tests

The explanations in this chapter make use of *litmus tests*, or small programs designed to test or highlight one particular aspect of a memory model. Figure A.1 shows an example of a litmus test with two harts. As a convention for this figure and for all figures that follow in this chapter, we assume that `s0`–`s2` are pre-set to the same value in all harts and that `s0` holds the address labeled `x`, `s1` holds `y`, and `s2` holds `z`, where `x`, `y`, and `z` are disjoint memory locations aligned to 8 byte boundaries. Each figure shows the litmus test code on the left, and a visualization of one particular valid or invalid execution on the right.

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)`
	⋮		⋮

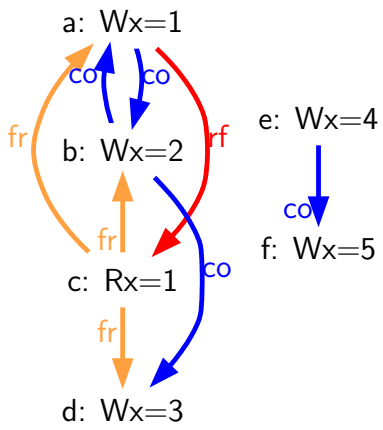


Figure A.1: A sample litmus test and one forbidden execution (`a0=1`).

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

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

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

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

notation is explained in Table A.1. Of the listed relations, rf edges between harts, co edges, fr edges, and ppo edges directly constrain the global memory order (as do fence, addr, data, and some ctrl edges, via ppo). Other edges (such as intra-hart rf edges) are informative but do not constrain the global memory order.

For example, in Figure A.1, `a0=1` could occur only if one of the following were true:

- (b) appears before (a) in global memory order (and in the coherence order co). However, this violates RVWMO PPO rule 1. The co edge from (b) to (a) highlights this contradiction.
- (a) appears before (b) in global memory order (and in the coherence order co). However, in this case, the Load Value Axiom would be violated, because (a) is not the latest matching store prior to (c) in program order. The fr edge from (c) to (b) highlights this contradiction.

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

Beyond what is described in this appendix, a suite of more than seven thousand litmus tests is available at http://diy.inria.fr/cats7/riscv/.

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

A.3 Explaining the RVWMO Rules

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

A.3.1 Preserved Program Order and Global Memory Order

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

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

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

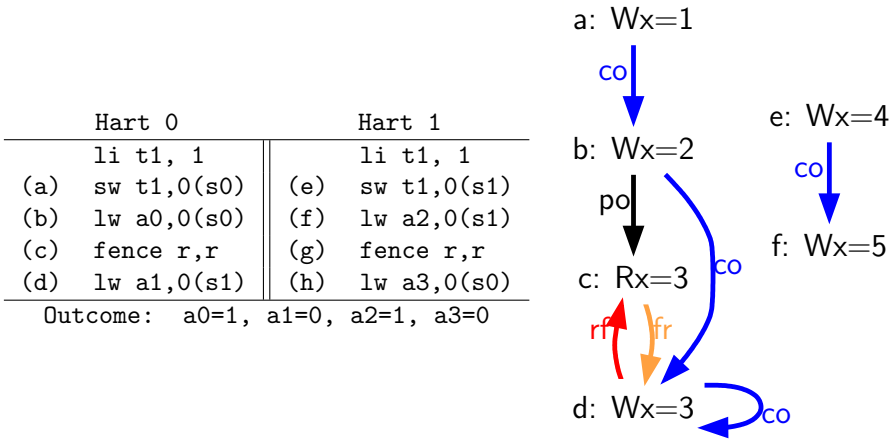
A.3.2 Load Value Axiom

> Load Value Axiom: Each byte of each load i returns the value written to that byte by the store that is the latest in global memory order among the following stores:
>
> 1. Stores that write that byte and that precede i in the global memory order
> 2. Stores that write that byte and that precede i in program order

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

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

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



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

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

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

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

The global memory order must be a total order and cannot be cyclic, because a cycle would imply that every event in the cycle happens before itself, which is impossible. Therefore, the execution

proposed above would be forbidden, and hence the addition of rule X would forbid implementations with store buffer forwarding, which would clearly be undesirable.

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

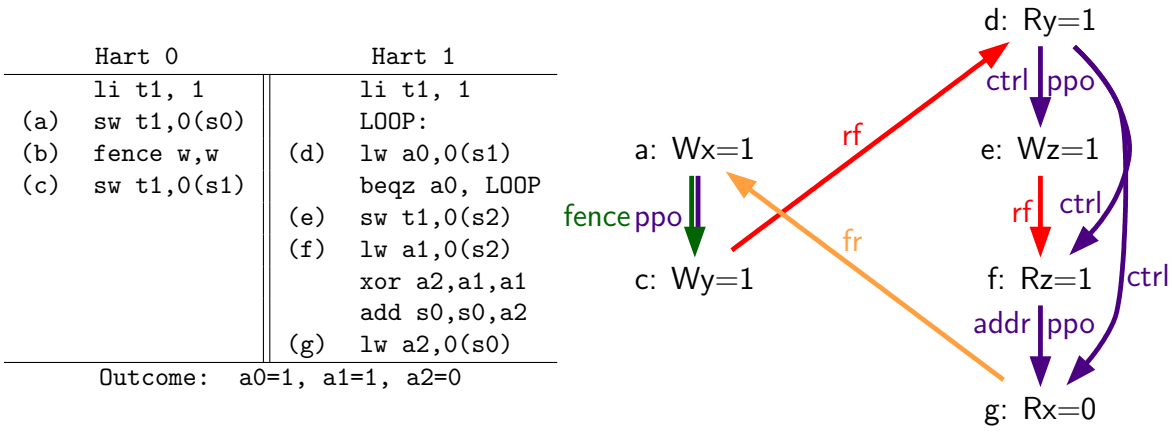


Figure A.3: The "PPOCA" store buffer forwarding litmus test (outcome permitted)

Another test that highlights the behavior of store buffers is shown in Figure A.3. In this example, (d) is ordered before (e) because of the control dependency, and (f) is ordered before (g) because of the address dependency. However, (e) is *not* necessarily ordered before (f), even though (f) returns the value written by (e). This could correspond to the following sequence of events:

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

A.3.3 Atomicity Axiom

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

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

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

```
(a) lr.d a0, 0(s0)     (a) lr.d a0, 0(s0)     (a) lr.w a0, 0(s0)     (a) lr.w a0, 0(s0)
(b) sd   t1, 0(s0)     (b) sw   t1, 4(s0)     (b) sw   t1, 4(s0)     (b) sw   t1, 4(s0)
(c) sc.d t2, 0(s0)     (c) sc.d t2, 0(s0)     (c) sc.w t2, 0(s0)     (c) sc.w t2, 8(s0)
```

Figure A.4: In all four (independent) code snippets, the store-conditional (c) is permitted but not guaranteed to succeed

The atomicity axiom forbids from other harts from being interleaved in global memory order between an LR and the SC paired with that LR. The atomicity axiom does not forbid loads from being interleaved between the paired operations in program order or in the global memory order, nor does it forbid stores from the same hart or stores to non-overlapping locations from appearing between the paired operations in either program order or in the global memory order. For example, the SC instructions in Figure A.4 may (but are not guaranteed to) succeed. None of those successes would violate the atomicity axiom, because the intervening non-conditional stores are from the same hart as the paired load-reserved and store-conditional instructions. This way, a memory system that tracks memory accesses at cache line granularity (and which therefore will see the four snippets of Figure A.4 as identical) will not be forced to fail a store conditional instruction that happens to (falsely) share another portion of the same cache line as the memory location being held by the reservation.

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

A.3.4 Progress Axiom

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

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

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

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

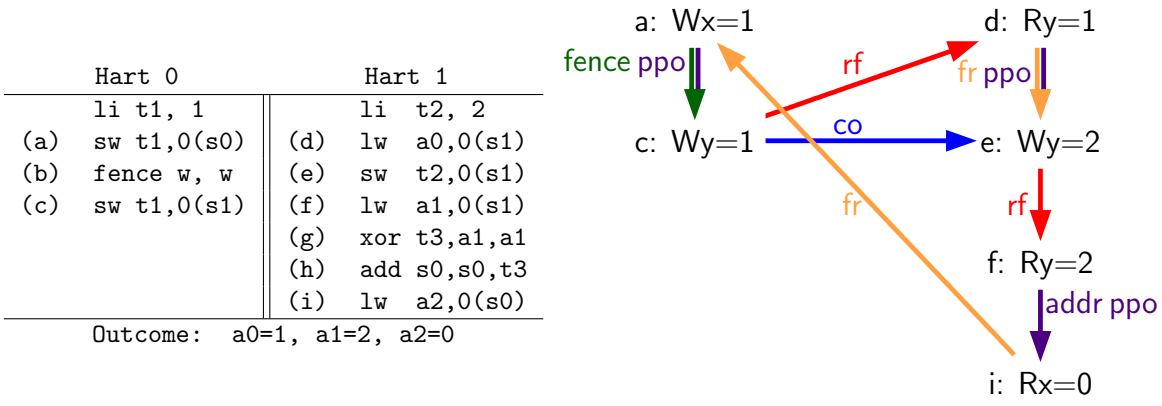
A.3.5 Overlapping-Address Orderings (Rules 1–3)

> Rule 1: b is a store, and a and b access overlapping memory addresses
> Rule 2: a and b are loads, x is a byte read by both a and b, there is no store to x between a and b in program order, and a and b return values for x written by different memory operations
> Rule 3: a is generated by an AMO or SC instruction, b is a load, and b returns a value written by a

Same-address orderings where the latter is a store are straightforward: a load or store can never be reordered with a later store to an overlapping memory location. From a microarchitecture perspective, generally speaking, it is difficult or impossible to undo a speculatively reordered store if the speculation turns out to be invalid, so such behavior is simply disallowed by the model. Same-address orderings from a store to a later load, on the other hand, do not need to be enforced. As discussed in Section A.3.2, this reflects the observable behavior of implementations that forward values from buffered stores to later loads.

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

Consider the litmus test of Figure A.5, which is one particular instance of the more general "fri-rfi" pattern. The term "fri-rfi" refers to the sequence (d), (e), (f): (d) "from-reads" (i.e., reads from



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

Outcome: a0=1, a1=2, a2=0

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

an earlier write than) (e) which is the same hart, and (f) reads from (e) which is in the same hart.

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

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

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

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

Consider the litmus test of Figure A.6. The outcome `a0=1`, `a1=`v, `a2=`v, `a3=0` (where v is some value written by another hart) can be observed by allowing (g) and (h) to be reordered. This might be done speculatively, and the speculation can be justified by the microarchitecture (e.g., by snooping for cache invalidations and finding none) because replaying (h) after (g) would return the value written by the same store anyway. Hence assuming `a1` and `a2` would end up with the same value

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

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

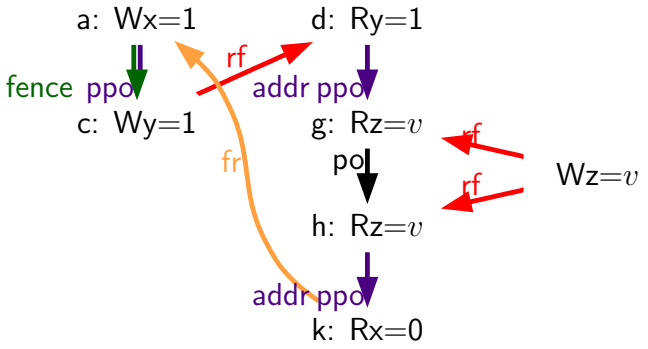


Figure A.6: Litmus test RSW (outcome permitted)

written by the same store anyway, (g) and (h) can be legally reordered. The global memory order corresponding to this execution would be (h),(k),(a),(c),(d),(g).

Executions of the test in Figure A.6 in which `a1` does not equal `a2` do in fact require that (g) appears before (h) in the global memory order. Allowing (h) to appear before (g) in the global memory order would in that case result in a violation of CoRR, because then (h) would return an older value than that returned by (g). Therefore, PPO rule 2 forbids this CoRR violation from occurring. As such, PPO rule 2 strikes a careful balance between enforcing CoRR in all cases while simultaneously being weak enough to permit "RSW" and "fri-rfi" patterns that commonly appear in real microarchitectures.

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

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

A.3.6 Fences (Rule 4)

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

By default, the FENCE instruction ensures that all memory accesses from instructions preceding the fence in program order (the "predecessor set") appear earlier in the global memory order than

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

The FENCE encoding currently has nine non-trivial combinations of the four bits PR, PW, SR, and SW, plus one extra encoding FENCE.TSO which facilitates mapping of "acquire+release" or RVTSO semantics. The remaining seven combinations have empty predecessor and/or successor sets and hence are no-ops. Of the ten non-trivial options, only six are commonly used in practice:

- FENCE RW,RW
- FENCE.TSO
- FENCE RW,W
- FENCE R,RW
- FENCE R,R
- FENCE W,W

FENCE instructions using any other combination of PR, PW, SR, and SW are reserved. We strongly recommend that programmers stick to these six. Other combinations may have unknown or unexpected interactions with the memory model.

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

A.3.7 Explicit Synchronization (Rules 5–8)

> Rule 5: a has an acquire annotation
> Rule 6: b has a release annotation
> Rule 7: a and b both have RCsc annotations
> Rule 8: a is paired with b

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

Consider Figure A.7. Because this example uses *aq*, the loads and stores in the critical section are guaranteed to appear in the global memory order after the AMOSWAP used to acquire the lock.

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

Figure A.7: A spinlock with atomics

However, assuming `a0`, `a1`, and `a2` point to different memory locations, the loads and stores in the critical section may or may not appear after the "Arbitrary unrelated load" at the beginning of the example in the global memory order.

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

Figure A.8: A spinlock with fences

Now, consider the alternative in Figure A.8. In this case, even though the AMOSWAP does not enforce ordering with an *aq* bit, the fence nevertheless enforces that the acquire AMOSWAP appears earlier in the global memory order than all loads and stores in the critical section. Note, however, that in this case, the fence also enforces additional orderings: it also requires that the "Arbitrary unrelated load" at the start of the program 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 "Arbitrary unrelated store" at the start of the snippet.) In this way, fence-enforced orderings are slightly coarser than orderings enforced by *.aq*.

Release orderings work exactly the same as acquire orderings, just in the opposite direction. Release semantics require all loads and stores preceding the release operation in program order to also precede the release operation in the global memory order. This ensures, for example, that memory

accesses in a critical section appear before the lock-releasing store in the global memory order. Just as for acquire semantics, release semantics can be enforced using release annotations or with a FENCE R,RW operations. Using the same examples, the ordering between the loads and stores in the critical section and the "Arbitrary unrelated store" at the end of the code snippet is enforced only by the FENCE R,RW in Figure A.8, not by the *rl* in Figure A.7.

With RCpc annotations alone, store-release-to-load-acquire ordering is not enforced. This facilitates the porting of code written under the TSO and/or RCpc memory models. To enforce store-release-to-load-acquire ordering, the code must use store-release-RCsc and load-acquire-RCsc operations so that PPO rule 7 applies. RCpc alone is sufficient for many uses cases in C/C++ but is insufficient for many other use cases in C/C++, Java, and Linux, to name just a few examples; see Section A.5 for details.

PPO rule 8 indicates that an SC must appear after its paired LR in the global memory order. This will follow naturally from the common use of LR/SC to perform an atomic read-modify-write operation due to the inherent data dependency. However, PPO rule 8 also applies even when the value being stored does not syntactically depend on the value returned by the paired LR.

Lastly, we note that just as with fences, programmers need not worry about "cumulativity" when analyzing ordering annotations.

A.3.8 Syntactic Dependencies (Rules 9–11)

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

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

The terms in Section 6.1 work as follows. Instructions are said to carry dependencies from their source register(s) to their destination register(s) whenever the value written into each destination register is a function of the source register(s). For most instructions, this means that the destination register(s) carry a dependency from all source register(s). However, there are a few notable exceptions. In the case of memory instructions, the value written into the destination register ultimately comes from the memory system rather than from the source register(s) directly, and so this breaks the chain of dependencies carried from the source register(s). In the case of unconditional jumps, the value written into the destination register comes from the current `pc` (which is never considered a source register by the memory model), and so likewise, JALR (the only jump with a source register) does not carry a dependency from *rs1* to *rd*.

The notion of accumulating into a destination register rather than writing into it reflects the behavior of CSRs such as `fflags`. In particular, an accumulation into a register does not clobber any previous writes or accumulations into the same register. For example, in Figure A.9, (c) has a

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

Figure A.9: (c) has a syntactic dependency on both (a) and (b) via `fflags`, a destination register that both (a) and (b) implicitly accumulate into

syntactic dependency on both (a) and (b).

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

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

Figure A.10: A syntactic address dependency

For example, there is a syntactic address dependency from the memory operation generated by the first instruction to the memory operation generated by the last instruction in Figure A.10, even though `a1` XOR `a1` is zero and hence has no effect on the address accessed by the second load.

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

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

Figure A.11: A syntactic control dependency

Control dependencies behave differently from address and data dependencies in the sense that a control dependency always extends to all instructions following the original target in program order. Consider Figure A.11: the instruction at `next` will always execute, but memory operation generated by that last instruction nevertheless still has control dependency from the memory operation generated by the first instruction.

Likewise, consider Figure A.12. Even though both branch outcomes have the same target, there is still a control dependency from the memory operation generated by the first instruction in this snippet to the memory operation generated by the last instruction. This definition of control

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

Figure A.12: Another syntactic control dependency

dependency is subtly stronger than what might be seen in other contexts (e.g., C++), but it conforms with standard definitions of control dependencies in the literature.

Notably, PPO rules 9–11 are also intentionally designed to respect dependencies that originate from the output of a successful store conditional instruction. Typically, an SC instruction will be followed by a conditional branch checking whether the outcome was successful; this implies that there will be a control dependency from the store operation generated by the SC instruction to any memory operations following the branch. PPO rule 11 in turn implies that any subsequent store operations will appear later in the global memory order than the store operation generated by the SC. However, since control, address, and data dependencies are defined over memory operations, and since an unsuccessful SC does not generate a memory operation, no order is enforced between unsuccessful SC and its dependent instructions. Moreover, since SC is defined to carry dependencies from its source registers to *rd* only when the SC is successful, an unsuccessful SC has no effect on the global memory order.

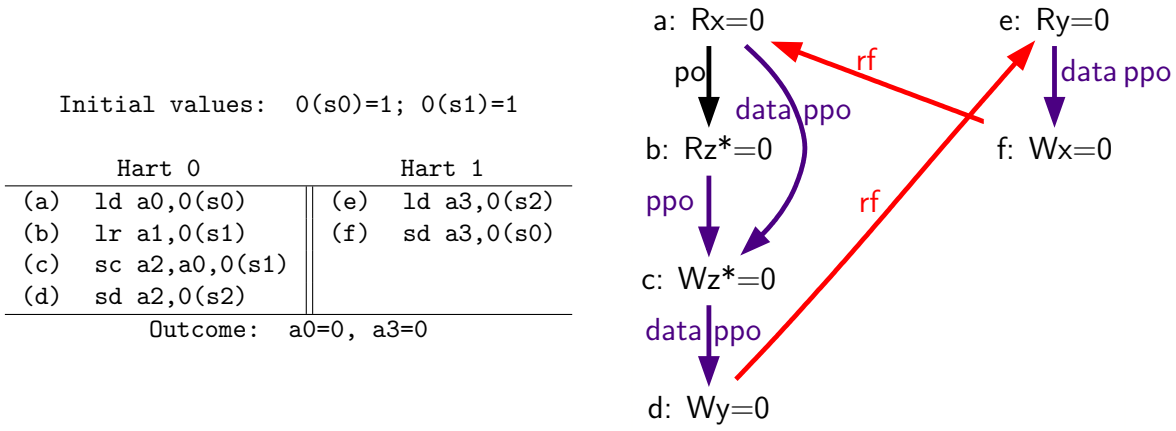


Figure A.13: A variant of the LB litmus test (outcome forbidden)

In addition, the choice to respect dependencies originating at store-conditional instructions ensures that certain out-of-thin-air-like behaviors will be prevented. Consider Figure A.13. Suppose a hypothetical implementation could occasionally make some early guarantee that a store-conditional operation will succeed. In this case, (c) could return 0 to `a2` early (before actually executing), allowing the sequence (d), (e), (f), (a), and then (b) to execute, and then (c) might execute (successfully) only at that point. This would imply that (c) writes its own success value to `0(s1)`! Fortunately, this situation and others like it are prevented by the fact that RVWMO respects dependencies originating at the stores generated by successful SC instructions.

We also note that syntactic dependencies between instructions only have any force when they take the form of a syntactic address, control, and/or data dependency. For example: a syntactic dependency between two "F" instructions via one of the "accumulating CSRs" in Section 6.3 does

not imply that the two "F" instructions must be executed in order. Such a dependency would only serve to ultimately set up later a dependency from both "F" instructions to a later CSR instruction accessing the CSR flag in question.

A.3.9 Pipeline Dependencies (Rules 12–13)

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

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

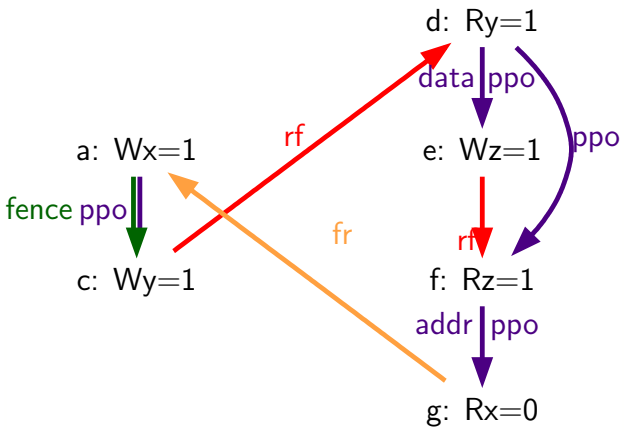


Figure A.14: Because of PPO rule 12 and the data dependency from (d) to (e), (d) must also precede (f) in the global memory order (outcome forbidden)

PPO rules 12 and 13 reflect behaviors of almost all real processor pipeline implementations. Rule 12 states that a load cannot forward from a store until the address and data for that store are known. Consider Figure A.14: (f) cannot be executed until the data for (e) has been resolved, because (f) must return the value written by (e) (or by something even later in the global memory order), and the old value must not be clobbered by the writeback of (e) before (d) has had a chance to perform. Therefore, (f) will never perform before (d) has performed.

If there were another store to the same address in between (e) and (f), as in Figure A.15, then (f) would no longer be dependent on the data of (e) being resolved, and hence the dependency of (f) on (d), which produces the data for (e), would be broken.

Rule 13 makes a similar observation to the previous rule: a store cannot be performed at memory until all previous loads that might access the same address have themselves been performed. Such a load must appear to execute before the store, but it cannot do so if the store were to overwrite the value in memory before the load had a chance to read the old value. Likewise, a store generally cannot be performed until it is known that preceding instructions will not cause an exception due to failed address resolution, and in this sense, rule 13 can be seen as somewhat of a special case of rule 11.

Consider Figure A.16: (f) cannot be executed until the address for (e) is resolved, because it may

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

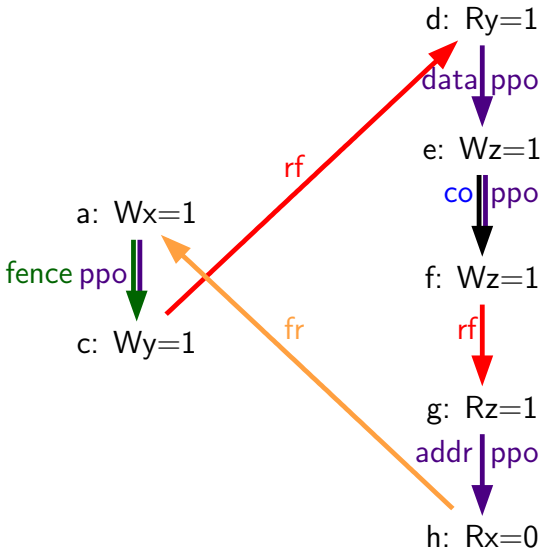


Figure A.15: Because of the extra store between (e) and (g), (d) no longer necessarily precedes (g) (outcome permitted)

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

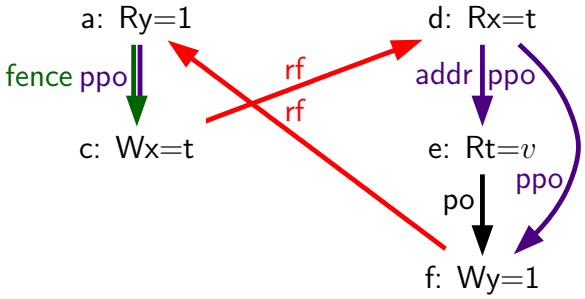


Figure A.16: Because of the address dependency from (d) to (e), (d) also precedes (f) (outcome forbidden)

turn out that the addresses match; i.e., that `a1=s0`. Therefore, (f) cannot be sent to memory before (d) has executed and confirmed whether the addresses do indeed overlap.

A.4 Beyond Main Memory

RVWMO does not currently attempt to formally describe how FENCE.I, SFENCE.VMA, I/O fences, and PMAs behave. All of these behaviors will be described by future formalizations. In the meantime, the behavior of FENCE.I is described in Section 2.9, the behavior of SFENCE.VMA is described in the RISC-V Instruction Set Privileged Architecture Manual, and the behavior of I/O fences and the effects of PMAs are described below.

A.4.1 Coherence and Cacheability

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

- Main memory vs. I/O, and I/O memory ordering PMAs: the memory model as defined applies to main memory regions. I/O ordering is discussed below.
- Supported access types and atomicity PMAs: the memory model is simply applied on top of whatever primitives each region supports.
- Cacheability PMAs: the cacheability PMAs in general do not affect the memory model. Non-cacheable regions may have more restrictive behavior than cacheable regions, but the set of allowed behaviors does not change regardless. However, some platform-specific and/or device-specific cacheability settings may differ.
- Coherence PMAs: The memory consistency model for memory regions marked as non-coherent in PMAs is currently platform-specific and/or device-specific: the load-value axiom, the atomicity axiom, and the progress axiom all may be violated with non-coherent memory. Note however that coherent memory does not require a hardware cache coherence protocol. The RISC-V Privileged ISA Specification suggests that hardware-incoherent regions of main memory are discouraged, but the memory model is compatible with hardware coherence, software coherence, implicit coherence due to read-only memory, implicit coherence due to only one agent having access, or otherwise.
- Idempotency PMAs: Idempotency PMAs are used to specify memory regions for which loads and/or stores may have side effects, and this in turn is used by the microarchitecture to determine, e.g., whether prefetches are legal. This distinction does not affect the memory model.

A.4.2 I/O Ordering

For I/O, the load value axiom and atomicity axiom in general do not apply, as both reads and writes might have device-specific side effects and may return values other than the value "written" by the most recent store to the same address. Nevertheless, the following preserved program order rules still generally apply for accesses to I/O memory: memory access a precedes memory access b in global memory order if a precedes b in program order and one or more of the following holds:

1. a precedes b in preserved program order as defined in Chapter 6, with the exception that acquire and release ordering annotations apply only from one memory operation to another memory operation and from one I/O operation to another I/O operation, but not from a memory operation to an I/O nor vice versa
2. a and b are accesses to overlapping addresses in an I/O region
3. a and b are accesses to the same strongly-ordered I/O region

4. a and b are accesses to I/O regions, and the channel associated with the I/O region accessed by either a or b is channel 1

5. a and b are accesses to I/O regions associated with the same channel (except for channel 0)

Note that the FENCE instruction distinguishes between main memory operations and I/O operations in its predecessor and successor sets. To enforce ordering between I/O operations and main memory operations, code must use a FENCE with PI, PO, SI, and/or SO, plus PR, PW, SR, and/or SW. For example, to enforce ordering between a write to main memory and an I/O write to a device register, a FENCE W,O or stronger is needed.

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

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

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

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

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

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

The ordering guarantees in this section may not apply beyond a platform-specific boundary between the RISC-V cores and the device. In particular, I/O accesses sent across an external bus (e.g., PCIe)

may be reordered before they reach their ultimate destination. Ordering must be enforced in such situations according to the platform-specific rules of those external devices and buses.

A.5 Code Porting and Mapping Guidelines

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

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

Table A.2 provides a mapping from TSO memory operations onto RISC-V memory instructions. Normal x86 loads and stores are all inherently acquire-RCpc and release-RCpc operations: TSO enforces all load-load, load-store, and store-store ordering by default. Therefore, under RVWMO, all TSO loads must be mapped onto a load followed by FENCE R,RW, and all TSO stores must be mapped onto FENCE RW,W followed by a store. TSO atomic read-modify-writes and x86 instructions using the LOCK prefix are fully-ordered and can be implemented either via an AMO with both *aq* and *rl* set, or via an LR with *aq* set, the arithmetic operation in question, an SC with both *aq* and *rl* set, and a conditional branch checking the success condition. In the latter case, the *rl* annotation on the LR turns out (for non-obvious reasons) to be redundant and can be omitted.

Alternatives to Table A.2 are also possible. A TSO store can be mapped onto AMOSWAP with *rl* set. However, since RVWMO PPO Rule 3 forbids forwarding of values from AMOs to subsequent loads, the use of AMOSWAP for stores may negatively affect performance. A TSO load can be mapped using LR with *aq* set: all such LR instructions will be unpaired, but that fact in and of itself does not preclude the use of LR for loads. However, again, this mapping may also negatively affect performance if it puts more pressure on the reservation mechanism than was originally intended.

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

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

Table A.3 provides a mapping from Power memory operations onto RISC-V memory instructions. Power ISYNC maps on RISC-V to a FENCE.I followed by a FENCE R,R; the latter fence is needed because ISYNC is used to define a "control+control fence" dependency that is not present in RVWMO.

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

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

Table A.4 provides a mapping from ARM memory operations onto RISC-V memory instructions. Since RISC-V does not currently have plain load and store opcodes with *aq* or *rl* annotations, ARM load-acquire and store-release operations should be mapped using fences instead. Furthermore, in order to enforce store-release-to-load-acquire ordering, there must be a FENCE RW,RW between the store-release and load-acquire; Table A.4 enforces this by always placing the fence in front of each acquire operation. ARM load-exclusive and store-exclusive instructions can likewise map onto their RISC-V LR and SC equivalents, but instead of placing a FENCE RW,RW in front of an LR with *aq* set, we simply also set *rl* instead. ARM ISB maps on RISC-V to FENCE.I followed by FENCE R,R similarly to how ISYNC maps for Power.

Table A.5 provides a mapping of Linux memory ordering macros onto RISC-V memory instructions. The Linux fences `dma_rmb()` and `dma_wmb()` map onto FENCE R,R and FENCE W,W, respectively, since the RISC-V Unix Platform requires coherent DMA, but would be mapped onto FENCE RI,RI and FENCE WO,WO, respectively, on a platform with non-coherent DMA. Platforms with non-coherent DMA may also require a mechanism by which cache lines can be flushed and/or invalidated. Such mechanisms will be device-specific and/or standardized in a future extension to the ISA.

The Linux mappings for release operations may seem stronger than necessary, but these mappings are needed to cover some cases in which Linux requires stronger orderings than the more intuitive mappings would provide. In particular, as of the time this text is being written, Linux is actively debating whether to require load-load, load-store, and store-store orderings between accesses in one critical section and accesses in a subsequent critical section in the same hart and protected by the same synchronization object. Not all combinations of FENCE RW,W/FENCE R,RW mappings with *aq*/*rl* mappings combine to provide such orderings. There are a few ways around this problem, including:

1. Always use FENCE RW,W/FENCE R,RW, and never use *aq*/*rl*. This suffices but is undesirable, as it defeats the purpose of the *aq*/*rl* modifiers.

2. Always use *aq*/*rl*, and never use FENCE RW,W/FENCE R,RW. This does not currently work due to the lack of load and store opcodes with *aq* and *rl* modifiers.

Linux Operation	RVWMO Mapping
smp_mb()	fence rw,rw
smp_rmb()	fence r,r
smp_wmb()	fence w,w
dma_rmb()	fence r,r
dma_wmb()	fence w,w
mb()	fence iorw,iorw
rmb()	fence ri,ri
wmb()	fence wo,wo
smp_load_acquire()	l{b|h|w|d}; fence r,rw
smp_store_release()	fence.tso; s{b|h|w|d}
Linux Construct	**RVWMO AMO Mapping**
atomic_<op>_relaxed	amo<op>.{w|d}
atomic_<op>_acquire	amo<op>.{w|d}.aq
atomic_<op>_release	amo<op>.{w|d}.rl
atomic_<op>	amo<op>.{w|d}.aqrl
Linux Construct	**RVWMO LR/SC Mapping**
atomic_<op>_relaxed	loop: lr.{w|d}; <op>; sc.{w|d}; bnez loop
atomic_<op>_acquire	loop: lr.{w|d}.aq; <op>; sc.{w|d}; bnez loop
atomic_<op>_release	loop: lr.{w|d}; <op>; sc.{w|d}.aqrl*; bnez loop OR fence.tso; loop: lr.{w|d}; <op>; sc.{w|d}*; bnez loop
atomic_<op>	loop: lr.{w|d}.aq; <op>; sc.{w|d}.aqrl; bnez loop

Table A.5: Mappings from Linux memory primitives to RISC-V primitives. Other constructs (such as spinlocks) should follow accordingly. Platforms or devices with non-coherent DMA may need additional synchronization (such as cache flush or invalidate mechanisms); currently any such extra synchronization will be device-specific.

3. Strengthen the mappings of release operations such that they would enforce sufficient orderings in the presence of either type of acquire mapping. This is the currently-recommended solution, and the one shown in Table A.5.

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

Figure A.18: Orderings between critical sections in Linux

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

change as the Linux Kernel Memory Model evolves.

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

Table A.6: Mappings from C/C++ primitives to RISC-V primitives.

Table A.6 provides a mapping of C11/C++11 atomic operations onto RISC-V memory instructions. If load and store opcodes with *aq* and *rl* modifiers are introduced, then the mappings in Table A.7 will suffice. Note however that the two mappings only interoperate correctly if `atomic_<op>(memory_order_seq_cst)` is mapped using an LR that has both *aq* and *rl* set.

Any AMO can be emulated by an LR/SC pair, but care must be taken to ensure that any PPO orderings that originate from the LR are also made to originate from the SC, and that any PPO orderings that terminate at the SC are also made to terminate at the LR. For example, the LR must also be made to respect any data dependencies that the AMO has, given that load operations do not otherwise have any notion of a data dependency. Likewise, the effect a FENCE R,R elsewhere in the same hart must also be made to apply to the SC, which would not otherwise respect that fence. The emulator may achieve this effect by simply mapping AMOs onto `lr.aq; <op>; sc.aqrl`,

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

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

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

matching the mapping used elsewhere for fully-ordered atomics.

A.6 Implementation Guidelines

The RVWMO and RVTSO memory models by no means preclude microarchitectures from employing sophisticated speculation techniques or other forms of optimization in order to deliver higher performance. The models also do not impose any requirement to use any one particular cache hierarchy, nor even to use a cache coherence protocol at all. Instead, these models only specify the behaviors that can be exposed to software. Microarchitectures are free to use any pipeline design, any coherent or non-coherent cache hierarchy, any on-chip interconnect, etc., as long as the design only admits executions that satisfy the memory model rules. That said, to help people understand the actual implementations of the memory model, in this section we provide some guidelines on how architects and programmers should interpret the models' rules.

Both RVWMO and RVTSO are multi-copy atomic (or "other-multi-copy-atomic"): any store value that is visible to a hart other than the one that originally issued it must also be conceptually visible to all other harts in the system. In other words, harts may forward from their own previous stores before those stores have become globally visible to all harts, but no early inter-hart forwarding is permitted. Multi-copy atomicity may be enforced in a number of ways. It might hold inherently due to the physical design of the caches and store buffers, it may be enforced via a single-writer/multiple-reader cache coherence protocol, or it might hold due to some other mechanism.

Although multi-copy atomicity does impose some restrictions on the microarchitecture, it is one of the key properties keeping the memory model from becoming extremely complicated. For example, a hart may not legally forward a value from a neighbor hart's private store buffer (unless of course it is done in such a way that no new illegal behaviors become architecturally visible). Nor may a cache coherence protocol forward a value from one hart to another until the coherence protocol has invalidated all older copies from other caches. Of course, microarchitectures may (and high-performance implementations likely will) violate these rules under the covers through speculation or other optimizations, as long as any non-compliant behaviors are not exposed to the programmer.

As a rough guideline for interpreting the PPO rules in RVWMO, we expect the following from the software perspective:

- programmers will use PPO rules 1 and 4–8 regularly and actively.
- expert programmers will use PPO rules 9–11 to speed up critical paths of important data structures.
- even expert programmers will rarely if ever use PPO rules 2–3 and 12–13 directly. These are included to facilitate common microarchitectural optimizations (rule 2) and the operational formal modeling approach (rules 3 and 12–13) described in Section B.3. They also facilitate the process of porting code from other architectures that have similar rules.

We also expect the following from the hardware perspective:

- PPO rules 1 and 3–6 reflect well-understood rules that should pose few surprises to architects.
- PPO rule 2 reflects a natural and common hardware optimization, but one that is very subtle and hence is worth double checking carefully.
- PPO rule 7 may not be immediately obvious to architects, but it is a standard memory model requirement
- The load value axiom, the atomicity axiom, and PPO rules 8–13 reflect rules that most hardware implementations will enforce naturally, unless they contain extreme optimizations. Of course, implementations should make sure to double check these rules nevertheless. Hardware must also ensure that syntactic dependencies are not "optimized away".

Architectures are free to implement any of the memory model rules as conservatively as they choose. For example, a hardware implementation may choose to do any or all of the following:

- interpret all fences as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), regardless of the bits actually set

- implement all fences with PW and SR as if they were FENCE RW,RW (or FENCE IORW,IORW, if I/O is involved), as PW with SR is the most expensive of the four possible main memory ordering components anyway
- emulate *aq* and *rl* as described in Section A.5
- enforcing all same-address load-load ordering, even in the presence of patterns such as "fri-rfi" and "RSW"
- forbid any forwarding of a value from a store in the store buffer to a subsequent AMO or LR to the same address
- forbid any forwarding of a value from an AMO or SC in the store buffer to a subsequent load to the same address
- implement TSO on all memory accesses, and ignore any main memory fences that do not include PW and SR ordering (e.g., as Ztso implementations will do)
- implement all atomics to be RCsc or even fully-ordered, regardless of annotation

Architectures that implement RVTSO can safely do the following:

- Ignore all fences that do not have both PW and SR (unless the fence also orders I/O)
- Ignore all PPO rules except for rules 4 through 7, since the rest are redundant with other PPO rules under RVTSO assumptions

Other general notes:

- Silent stores (i.e., stores that write the same value that already exists at a memory location) behave like any other store from a memory model point of view. Likewise, AMOs which do not actually change the value in memory (e.g., an AMOMAX for which the value in *rs2* is smaller than the value currently in memory) are still semantically considered store operations. Microarchitectures that attempt to implement silent stores must take care to ensure that the memory model is still obeyed, particularly in cases such as RSW (Section A.3.5) which tend to be incompatible with silent stores.
- Writes may be merged (i.e., two consecutive writes to the same address may be merged) or subsumed (i.e., the earlier of two back-to-back writes to the same address may be elided) as long as the resulting behavior does not otherwise violate the memory model semantics.

The question of write subsumption can be understood from the following example:

As written, if the load (d) reads value 1, then (a) must precede (f) in the global memory order:

- (a) precedes (c) in the global memory order because of rule 2
- (c) precedes (d) in the global memory order because of the Load Value axiom

	Hart 0		Hart 1
	`li t1, 3`		`li t3, 2`
	`li t2, 1`		
(a)	`sw t1,0(s0)`	(d)	`lw a0,0(s1)`
(b)	`fence w, w`	(e)	`sw a0,0(s0)`
(c)	`sw t2,0(s1)`	(f)	`sw t3,0(s0)`

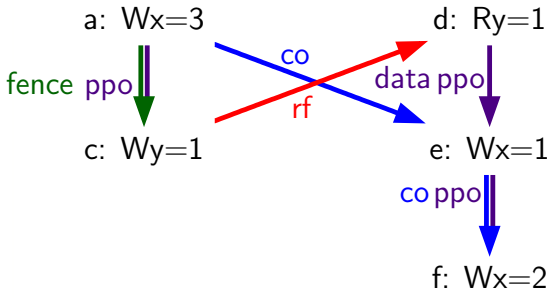


Figure A.19: Write subsumption litmus test, allowed execution.

- (d) precedes (e) in the global memory order because of rule 7
- (e) precedes (f) in the global memory order because of rule 1

In other words the final value of the memory location whose address is in `s0` must be 2 (the value written by the store (f)) and cannot be 3 (the value written by the store (a)).

A very aggressive microarchitecture might erroneously decide to discard (e), as (f) supersedes it, and this may in turn lead the microarchitecture to break the now-eliminated dependency between (d) and (f) (and hence also between (a) and (f)). This would violate the memory model rules, and hence it is forbidden. Write subsumption may in other cases be legal, if for example there were no data dependency between (d) and (e).

A.6.1 Possible Future Extensions

We expect that any or all of the following possible future extensions would be compatible with the RVWMO memory model:

- 'V' vector ISA extensions
- A transactional memory subset of the 'T' ISA extension
- 'J' JIT extension
- Native encodings for load and store opcodes with *aq* and *rl* set
- Fences limited to certain addresses
- Cache writeback/flush/invalidate/etc. instructions

	Hart 0		Hart 1
	li t1, 1		li t1, 1
(a)	lw a0,0(s0)	(d)	lw a1,0(s1)
(b)	fence rw,rw	(e)	amoswap.w.rl a2,t1,0(s2)
(c)	sw t1,0(s1)	(f)	ld a3,0(s2)
		(g)	lw a4,4(s2)
			xor a5,a4,a4
			add s0,s0,a5
		(h)	sw a2,0(s0)
Outcome: a0=1, a1=1, a2=0, a3=1, a4=0			

Figure A.20: Mixed-size discrepancy (permitted by axiomatic models, forbidden by operational model)

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

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

	Hart 0		Hart 1
	li t1, 1		li t1, 1
(a)	lw a0,0(s0)	(d)	sw t1,4(s1)
(b)	fence rw,rw	(e)	ld a1,0(s1)
(c)	sw t1,0(s1)	(f)	lw a2,4(s1)
			xor a3,a2,a2
			add s0,s0,a3
		(g)	sw a2,0(s0)
Outcome: a0=1, a1=0x100000001, a1=1			

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

A.7 Known Issues

A.7.1 Mixed-size RSW

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

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

Appendix B

Formal Memory Model Specifications, Version 0.1

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

This appendix should be treated as commentary; all normative material is provided in Chapter 6 and in the rest of the main body of the ISA specification. All currently known discrepancies are listed in Section A.7. Any other discrepancies are unintentional.

B.1 Formal Axiomatic Specification in Alloy

We present a formal specification of the RVWMO memory model in Alloy (http://alloy.mit.edu). This model is available online at https://github.com/daniellustig/riscv-memory-model.

The online material also contains some litmus tests and some examples of how Alloy can be used to model check some of the mappings in Section A.5.

```
////////////////////////////////////////////////////////////////////////////
// =RVWMO PPO=

// Preserved Program Order
fun ppo : Event->Event {
  // same-address ordering
  po_loc :> Store
  + rdw
  + (AMO + StoreConditional) <: rfi

  // explicit synchronization
  + ppo_fence
  + Acquire <: ^po :> MemoryEvent
  + MemoryEvent <: ^po :> Release
  + RCsc <: ^po :> RCsc
  + pair

  // syntactic dependencies
  + addrdep
  + datadep
  + ctrldep :> Store

  // pipeline dependencies
  + (addrdep+datadep).rfi
  + addrdep.^po :> Store
}

// the global memory order respects preserved program order
fact { ppo in ^gmo }
```

Figure B.1: The RVWMO memory model formalized in Alloy (1/5: PPO)

```
////////////////////////////////////////////////////////////////////////////////
// =RVWMO axioms=

// Load Value Axiom
fun candidates[r: MemoryEvent] : set MemoryEvent {
  (r.~^gmo & Store & same_addr[r]) // writes preceding r in gmo
  + (r.^~po & Store & same_addr[r]) // writes preceding r in po
}

fun latest_among[s: set Event] : Event { s - s.~^gmo }

pred LoadValue {
  all w: Store | all r: Load |
    w->r in rf <=> w = latest_among[candidates[r]]
}

// Atomicity Axiom
pred Atomicity {
  all r: Store.~pair |             // starting from the lr,
    no x: Store & same_addr[r] |  // there is no store x to the same addr
      x not in same_hart[r]       // such that x is from a different hart,
      and x in r.~rf.^gmo         // x follows (the store r reads from) in gmo,
      and r.pair in x.^gmo        // and r follows x in gmo
}

// Progress Axiom implicit: Alloy only considers finite executions

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

Figure B.2: The RVWMO memory model formalized in Alloy (2/5: Axioms)

```
////////////////////////////////////////////////////////////////////////////////
// Basic model of memory

sig Hart {  // hardware thread
  start : one Event
}
sig Address {}
abstract sig Event {
  po: lone Event // program order
}

abstract sig MemoryEvent extends Event {
  address: one Address,
  acquireRCpc: lone MemoryEvent,
  acquireRCsc: lone MemoryEvent,
  releaseRCpc: lone MemoryEvent,
  releaseRCsc: lone MemoryEvent,
  addrdep: set MemoryEvent,
  ctrldep: set Event,
  datadep: set MemoryEvent,
  gmo: set MemoryEvent,  // global memory order
  rf: set MemoryEvent
}
sig LoadNormal extends MemoryEvent {} // l{b|h|w|d}
sig LoadReserve extends MemoryEvent { // lr
  pair: lone StoreConditional
}
sig StoreNormal extends MemoryEvent {}       // s{b|h|w|d}
// all StoreConditionals in the model are assumed to be successful
sig StoreConditional extends MemoryEvent {}  // sc
sig AMO extends MemoryEvent {}               // amo
sig NOP extends Event {}

fun Load : Event { LoadNormal + LoadReserve + AMO }
fun Store : Event { StoreNormal + StoreConditional + AMO }

sig Fence extends Event {
  pr: lone Fence, // opcode bit
  pw: lone Fence, // opcode bit
  sr: lone Fence, // opcode bit
  sw: lone Fence  // opcode bit
}
sig FenceTSO extends Fence {}

/* Alloy encoding detail: opcode bits are either set (encoded, e.g.,
 * as f.pr in iden) or unset (f.pr not in iden).  The bits cannot be used for
 * anything else */
fact { pr + pw + sr + sw in iden }
// likewise for ordering annotations
fact { acquireRCpc + acquireRCsc + releaseRCpc + releaseRCsc in iden }
// don't try to encode FenceTSO via pr/pw/sr/sw; just use it as-is
fact { no FenceTSO.(pr + pw + sr + sw) }
```

Figure B.3: The RVWMO memory model formalized in Alloy (3/5: model of memory)

```
////////////////////////////////////////////////////////////////////////////////
// =Basic model rules=

// Ordering annotation groups
fun Acquire : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.acquireRCsc }
fun Release : MemoryEvent { MemoryEvent.releaseRCpc + MemoryEvent.releaseRCsc }
fun RCpc : MemoryEvent { MemoryEvent.acquireRCpc + MemoryEvent.releaseRCpc }
fun RCsc : MemoryEvent { MemoryEvent.acquireRCsc + MemoryEvent.releaseRCsc }

// There is no such thing as store-acquire or load-release, unless it's both
fact { Load & Release in Acquire }
fact { Store & Acquire in Release }

// FENCE PPO
fun FencePRSR : Fence { Fence.(pr & sr) }
fun FencePRSW : Fence { Fence.(pr & sw) }
fun FencePWSR : Fence { Fence.(pw & sr) }
fun FencePWSW : Fence { Fence.(pw & sw) }

fun ppo_fence : MemoryEvent->MemoryEvent {
    (Load  <: ^po :> FencePRSR).(^po :> Load)
  + (Load  <: ^po :> FencePRSW).(^po :> Store)
  + (Store <: ^po :> FencePWSR).(^po :> Load)
  + (Store <: ^po :> FencePWSW).(^po :> Store)
  + (Load  <: ^po :> FenceTSO) .(^po :> MemoryEvent)
  + (Store <: ^po :> FenceTSO) .(^po :> Store)
}

// auxiliary definitions
fun po_loc : Event->Event { ^po & address.~address }
fun same_hart[e: Event] : set Event { e + e.^~po + e.^po }
fun same_addr[e: Event] : set Event { e.address.~address }

// initial stores
fun NonInit : set Event { Hart.start.*po }
fun Init : set Event { Event - NonInit }
fact { Init in StoreNormal }
fact { Init->(MemoryEvent & NonInit) in ^gmo }
fact { all e: NonInit | one e.*~po.~start }  // each event is in exactly one hart
fact { all a: Address | one Init & a.~address } // one init store per address
fact { no Init <: po and no po :> Init }
```

Figure B.4: The RVWMO memory model formalized in Alloy (4/5: Basic model rules)

```
// po
fact { acyclic[po] }

// gmo
fact { total[^gmo, MemoryEvent] } // gmo is a total order over all MemoryEvents

//rf
fact { rf.~rf in iden } // each read returns the value of only one write
fact { rf in Store <: address.~address :> Load }
fun rfi : MemoryEvent->MemoryEvent { rf & (*po + *~po) }

//dep
fact { no StoreNormal <: (addrdep + ctrldep + datadep) }
fact { addrdep + ctrldep + datadep + pair in ^po }
fact { datadep in datadep :> Store }
fact { ctrldep.*po in ctrldep }
fact { no pair & (^po :> (LoadReserve + StoreConditional)).^po }
fact { StoreConditional in LoadReserve.pair } // assume all SCs succeed

// rdw
fun rdw : Event->Event {
  (Load <: po_loc :> Load)  // start with all same_address load-load pairs,
  - (~rf.rf)                // subtract pairs that read from the same store,
  - (po_loc.rfi)            // and subtract out "fri-rfi" patterns
}

// filter out redundant instances and/or visualizations
fact { no gmo & gmo.gmo } // keep the visualization uncluttered
fact { all a: Address | some a.~address }

////////////////////////////////////////////////////////////////////////////////
// =Optional: opcode encoding restrictions=

// the list of blessed fences
fact { Fence in
  Fence.pr.sr
  + Fence.pw.sw
  + Fence.pr.pw.sw
  + Fence.pr.sr.sw
  + FenceTSO
  + Fence.pr.pw.sr.sw
}

pred restrict_to_current_encodings {
  no (LoadNormal + StoreNormal) & (Acquire + Release)
}

////////////////////////////////////////////////////////////////////////////////
// =Alloy shortcuts=
pred acyclic[rel: Event->Event] { no iden & ^rel }
pred total[rel: Event->Event, bag: Event] {
  all disj e, e': bag | e->e' in rel + ~rel
  acyclic[rel]
}
```

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

B.2 Formal Axiomatic Specification in Herd

The tool herd takes a memory model and a litmus test as input and simulates the execution of the test on top of the memory model. Memory models are written in the domain specific language Cat. This section provides two Cat memory model of RVWMO. The first model, Figure B.7, follows the *global memory order*, Chapter 6, definition of RVWMO, as much as is possible for a Cat model. The second model, Figure B.8, is an equivalent, more efficient, partial order based RVWMO model.

The simulator herd is part of the diy tool suite — see http://diy.inria.fr for software and documentation. The models and more are available online at http://diy.inria.fr/cats7/riscv/.

```
(*************)
(* Utilities *)
(*************)

(* All fence relations *)
let fence.r.r = [R];fencerel(Fence.r.r);[R]
let fence.r.w = [R];fencerel(Fence.r.w);[W]
let fence.r.rw = [R];fencerel(Fence.r.rw);[M]
let fence.w.r = [W];fencerel(Fence.w.r);[R]
let fence.w.w = [W];fencerel(Fence.w.w);[W]
let fence.w.rw = [W];fencerel(Fence.w.rw);[M]
let fence.rw.r = [M];fencerel(Fence.rw.r);[R]
let fence.rw.w = [M];fencerel(Fence.rw.w);[W]
let fence.rw.rw = [M];fencerel(Fence.rw.rw);[M]
let fence.tso =
  let f = fencerel(Fence.tso) in
  ([W];f;[W]) | ([R];f;[M])

let fence =
  fence.r.r | fence.r.w | fence.r.rw |
  fence.w.r | fence.w.w | fence.w.rw |
  fence.rw.r | fence.rw.w | fence.rw.rw |
  fence.tso

(* Same address, no W to the same address in-between *)
let po-loc-no-w = po-loc \ (po-loc?;[W];po-loc)
(* Read same write *)
let rsw = rf^-1;rf
(* Acquire, or stronger  *)
let AQ = Acq|AcqRel
(* Release or stronger *)
and RL = RelAcqRel
(* All RCsc *)
let RCsc = Acq|Rel|AcqRel
(* Amo events are both R and W, relation rmw relates paired lr/sc *)
let AMO = R & W
let StCond = range(rmw)

(*************)
(* ppo rules *)
(*************)

(* Overlapping-Address Orderings *)
let r1 = [M];po-loc;[W]
and r2 = ([R];po-loc-no-w;[R]) \ rsw
and r3 = [AMO|StCond];rfi;[R]
(* Explicit Synchronization *)
and r4 = fence
and r5 = [AQ];po;[M]
and r6 = [M];po;[RL]
and r7 = [RCsc];po;[RCsc]
and r8 = rmw
(* Syntactic Dependencies *)
and r9 = [M];addr;[M]
and r10 = [M];data;[W]
and r11 = [M];ctrl;[W]
(* Pipeline Dependencies *)
and r12 = [R];(addr|data);[W];rfi;[R]
and r13 = [R];addr;[M];po;[W]

let ppo = r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 | r10 | r11 | r12 | r13
```

Figure B.6: `riscv-defs.cat`, a herd definition of preserved program order (1/3)

```
Total

(* Notice that herd has defined its own rf relation *)

(* Define ppo *)
include "riscv-defs.cat"

(*******************************)
(* Generate global memory order *)
(*******************************)

let gmo0 = (* precursor: ie build gmo as an total order that include gmo0 *)
  loc & (W\FW) * FW | # Final write after any write to the same location
  ppo |               # ppo compatible
  rfe                 # includes herd external rf (optimisation)

(* 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

(* Atomicity axiom *)
let infloc = (gmo & loc)^-1
let inflocext = infloc & ext
let winside  = (infloc;rmw;inflocext) & (infloc;rf;rmw;inflocext) & [W]
empty winside as Atomic
```

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`, an alternative herd presentation of the RVWMO memory model (3/3)

B.3 An Operational Memory Model

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behaviour as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavour: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from more implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behaviour of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

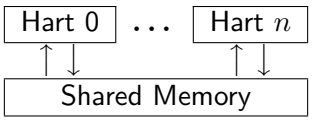
The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

An interactive version of the model, together with a library of litmus tests, is provided online: http://www.cl.cam.ac.uk/~pes20/rmem. This is integrated with a fragment of the RISC-V ISA semantics (RV64I and A) expressed explicitly in Sail (https://github.com/rems-project/sail)).

Below is an informal introduction of the model states and transitions. The description of the formal model starts in the next subsection.

Terminology: In contrast to the axiomatic presentation, here every memory operation is either a load or a store. Hence, AMOs give rise to two distinct memory operations, a load and a store. When used in conjunction with "instruction", the terms "load" and "store" refer to instructions that give rise to such memory operations. As such, both include AMO instructions. The term "acquire" refers to instruction (or its memory operation) with the acquire-RCpc or acquire-RCsc annotation. The term "release" refers to instruction (or its memory operation) with the release-RCpc or release-RCsc annotation.

Model states A model state consists of a shared memory and a tuple of hart states.



The shared memory state records all the memory store operations that have propagated so far, in the order they propagated (this can be made more efficient, but for simplicity of the presentation we keep it this way).

Each hart state consists principally of a tree of instruction instances, some of which have been *finished*, and some of which have not. Non-finished instruction instances can be subject to *restart*, e.g. if they depend on an out-of-order or speculative load that turns out to be unsound.

Conditional branch and indirect jump instructions may have multiple successors in the instruction tree. When such instruction is finished, any un-taken alternative paths are discarded.

Each instruction instance in the instruction tree has a state that includes an execution state of the intra-instruction semantics (the ISA pseudocode for this instruction). The model uses a formalisation of the intra-instruction semantics in Sail. One can think of the execution state of an instruction as a representation of the pseudocode control state, pseudocode call stack, and local variable values. An instruction instance state also includes information about the instance's memory and register footprints, its register reads and writes, its memory operations, whether it is finished, etc.

Model transitions The model defines, for any model state, the set of allowed transitions, each of which is a single atomic step to a new abstract machine state. Execution of a single instruction will typically involve many transitions, and they may be interleaved in operational-model execution with transitions arising from other instructions. Each transition arises from a single instruction instance; it will change the state of that instance, and it may depend on or change the rest of its hart state and the shared memory state, but it does not depend on other hart states, and it will not change them. The transitions are introduced below and defined in Section B.3.5, with a precondition and a construction of the post-transition model state for each.

Transitions for all instructions:

- Fetch instruction: This transition represents a fetch and decode of a new instruction instance, as a program order successor of a previously fetched instruction instance (or the initial fetch address).

  The model assumes the instruction memory is fixed; it does not describe the behaviour of self-modifying code. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.

- Register write: This is a write of a register value.
- Register read: This is a read of a register value from the most recent program-order-predecessor instruction instance that writes to that register.
- Pseudocode internal step: This covers pseudocode internal computation: arithmetic, function calls, etc.
- Finish instruction: At this point the instruction pseudocode is done, the instruction cannot be restarted, memory accesses cannot be discarded, and all memory effects have taken place. For conditional branch and indirect jump instructions, any program order successors that were fetched from an address that is not the one that was written to the *pc* register are discarded, together with the sub-tree of instruction instances below them.

Transitions specific to load instructions:

- Initiate memory load operations: At this point the memory footprint of the load instruction is provisionally known (it could change if earlier instructions are restarted) and its individual memory load operations can start being satisfied.

- Satisfy memory load operation by forwarding from unpropagated stores: This partially or entirely satisfies a single memory load operation by forwarding, from program-order-previous memory store operations.
- Satisfy memory load operation from memory: This entirely satisfies the outstanding slices of a single memory load operation, from memory.
- Complete load operations: At this point all the memory load operations of the instruction have been entirely satisfied and the instruction pseudocode can continue executing. A load instruction can be subject to being restarted until the Finish instruction transition. But, under some conditions, the model might treat a load instruction as non-restartable even before it is finished (e.g. see Propagate store operation).

Transitions specific to store instructions:

- Initiate memory store operation footprints: At this point the memory footprint of the store is provisionally known.
- Instantiate memory store operation values: At this point the memory store operations have their values and program-order-successor memory load operations can be satisfied by forwarding from them.
- Commit store instruction: At this point the store operations are guaranteed to happen (the instruction can no longer be restarted or discarded), and they can start being propagated to memory.
- Propagate store operation: This propagates a single memory store operation to memory.
- Complete store operations: At this point all the memory store operations of the instruction have been propagated to memory, and the instruction pseudocode can continue executing.

Transitions specific to `sc` instructions:

- Early `sc` fail: This causes the `sc` to fail, either a spontaneous fail or because it is not paired with a program-order-previous `lr`.
- Paired `sc`: This transition indicates the `sc` is paired with an `lr` and might succeed.
- Commit and propagate store operation of an `sc`: This is an atomic execution of the transitions Commit store instruction and Propagate store operation, it is enabled only if the stores from which the `lr` read from have not been overwritten.
- Late `sc` fail: This causes the `sc` to fail, either a spontaneous fail or because the stores from which the `lr` read from have been overwritten.

Transitions specific to AMO instructions:

- Satisfy, commit and propagate operations of an AMO: This is an atomic execution of all the transitions needed to satisfy the load operation, do the required arithmetic, and propagate the store operation.

Transitions specific to fence instructions:

- ∘ Commit fence

The transitions labelled ∘ can always be taken eagerly, as soon as their precondition is satisfied, without excluding other behaviour; the • cannot. Although Fetch instruction is marked with a •, it can be taken eagerly as long as it is not taken infinitely many times.

An instance of a non-AMO load instruction, after being fetched, will typically experience the following transitions in this order:

1. Register read
2. Initiate memory load operations
3. Satisfy memory load operation by forwarding from unpropagated stores and/or Satisfy memory load operation from memory (as many as needed to satisfy all the load operations of the instance)
4. Complete load operations
5. Register write
6. Finish instruction

Before, between and after the transitions above, any number of Pseudocode internal step transitions may appear. In addition, a Fetch instruction transition for fetching the instruction in the next program location will be available until it is taken.

This concludes the informal description of the operational model. The following sections describe the formal operational model.

B.3.1 Intra-instruction Pseudocode Execution

The intra-instruction semantics for each instruction instance is expressed as a state machine, essentially running the instruction pseudocode. Given a pseudocode execution state, it computes the next state. Most states identify a pending memory or register operation, requested by the pseudocode, which the memory model has to do. The states are (this is a tagged union; tags in small-caps):

LOAD_MEM(*kind*, *address*, *size*, *load_continuation*)	-	memory load operation
EARLY_SC_FAIL(*res_continuation*)	-	allow `sc` to fail early
STORE_EA(*kind*, *address*, *size*, *next_state*)	-	memory store effective address
STORE_MEMV(*mem_value*, *store_continuation*)	-	memory store value
FENCE(*kind*, *next_state*)	-	fence
READ_REG(*reg_name*, *read_continuation*)	-	register read
WRITE_REG(*reg_name*, *reg_value*, *next_state*)	-	register write
INTERNAL(*next_state*)	-	pseudocode internal step
DONE	-	end of pseudocode

Here:

- *mem_value* and *reg_value* are lists of bytes;
- *address* is an integer of XLEN bits;
- for load/store, *kind* identifies whether it is `lr/sc`, acquire-RCpc/release-RCpc, acquire-RCsc/release-RCsc, acquire-release-RCsc;
- for fence, *kind* identifies whether it is a normal or TSO, and (for normal fences) the predecessor and successor ordering bits;
- *reg_name* identifies a register and a slice thereof (start and end bit indices); and
- the continuations describe how the instruction instance will continue for each value that might be provided by the surrounding memory model (the *load_continuation* and *read_continuation* take the value loaded from memory and read from the previous register write, the *store_continuation* takes *false* for an `sc` that failed and *true* in all other cases, and *res_continuation* takes *false* if the `sc` fails and *true* otherwise).

For example, given the load instruction `lw x1,0(x2)`*, an execution will typically go as follows. The initial execution state will be computed from the pseudocode for the given opcode. This can be expected to be* READ_REG*(*`x2`*, read_continuation). Feeding the most recently written value of register* `x2` *(the instruction semantics will be blocked if necessary until the register value is available), say* `0x4000`*, to read_continuation returns* LOAD_MEM*(*`plain_load`*,* `0x4000`*,* `4`*, load_continuation). Feeding the 4-byte value loaded from memory location* `0x4000`*, say* `0x42`*, to load_continuation returns* WRITE_REG*(*`x1`*,* `0x42`*,* DONE*). Many* INTERNAL*(next_state) states may appear before and between the states above.*

Notice that writing to memory is split into two steps, STORE_EA and STORE_MEMV: the first one makes the memory footprint of the store provisionally known, and the second one adds the value to be stored. We ensure these are paired in the pseudocode (STORE_EA followed by STORE_MEMV), but there may be other steps between them.

It is observable that the STORE_EA *can occur before the value to be stored is determined. For example, for the litmus test LB+fence.r.rw+data-po to be allowed by the operational model (as it is by RVWMO), the first store in Hart 1 has to take the* STORE_EA *step before its value is determined, so that the second store can see it is to a non-overlapping memory footprint, allowing the second store to be committed out of order without violating coherence.*

The pseudocode of each instruction performs at most one store or one load, except for AMOs that perform exactly one load and one store. Those memory accesses are then split apart into the architecturally atomic units by the hart semantics (see Initiate memory load operations and Initiate memory store operation footprints below).

Informally, each bit of a register read should be satisfied from a register write by the most recent (in program order) instruction instance that can write that bit (or from the hart's initial register state if there is no such write). Hence, it is essential to know the register write footprint of each instruction instance, which we calculate when the instruction instance is created (see the action of Fetch instruction below). We ensure in the pseudocode that each instruction does at most one register write to each register bit, and also that it does not try to read a register value it just wrote.

Data-flow dependencies (address and data) in the model emerge from the fact that each register read has to wait for the appropriate register write to be executed (as described above).

B.3.2 Instruction Instance State

Each instruction instance i has a state comprising:

- *program_loc*, the memory address from which the instruction was fetched;
- *instruction_kind*, identifying whether this is a load, store, AMO, fence, branch/jump or a 'simple' instruction (this also includes a *kind* similar to the one described for the pseudocode execution states);
- *src_regs*, the set of source *reg_name*s (including system registers), as statically determined from the pseudocode of the instruction;
- *dst_regs*, the destination *reg_name*s (including system registers), as statically determined from the pseudocode of the instruction;
- *pseudocode_state* (or sometimes just 'state' for short), one of (this is a tagged union; tags in small-caps):

PLAIN(*isa_state*)	-	ready to make a pseudocode transition
PENDING_MEM_LOADS(*load_continuation*)	-	requesting memory load operation(s)
PENDING_MEM_STORES(*store_continuation*)	-	requesting memory store operation(s)

- *reg_reads*, the register reads the instance has performed, including, for each one, the register write slices it read from;
- *reg_writes*, the register writes the instance has performed;
- *mem_loads*, a set of memory load operations, and for each one the as-yet-unsatisfied slices (the byte indices that have not been satisfied yet), and, for the satisfied slices, the store slices (each consisting of a memory store operation and subset of its byte indices) that satisfied it.
- *mem_stores*, a set of memory store operations, and for each one a flag that indicates whether it has been propagated (passed to the shared memory) or not.
- information recording whether the instance is committed, finished, etc.

Each memory load operation includes a memory footprint (address and size). Each memory store operations includes a memory footprint, and, when available, a value.

A load instruction instance with a non-empty *mem_loads*, for which all the load operations are satisfied (i.e. there are no unsatisfied load slices) is said to be *entirely satisfied*.

Informally, an instruction instance is said to have *fully determined data* if the load (and `sc`) instructions feeding its source registers are finished. Similarly, it is said to have a *fully determined memory footprint* if the load (and `sc`) instructions feeding its memory operation address register are finished. Formally, we first define the notion of *fully determined register write*: a register write w from *reg_writes* of instruction instance i is said to be *fully determined* if one of the following conditions hold:

1. i is finished; or

2. the value written by w is not affected by a memory operation that i has made (i.e. a value loaded from memory or the result of `sc`), and, for every register read that i has made, that affects w, the register write from which i read is fully determined (or i read from the initial register state).

Now, an instruction instance i is said to have *fully determined data* if for every register read r from *reg_reads*, the register writes that r reads from are fully determined. An instruction instance i is said to have a *fully determined memory footprint* if for every register read r from *reg_reads* that feeds into i's memory operation address, the register writes that r reads from are fully determined.

The `rmem` *tool records, for every register write, the set of register writes from other instructions that have been read by this instruction at the point of performing the write. By carefully arranging the pseudocode of the instructions covered by the tool we were able to make it so that this is exactly the set of register writes on which the write depends on.*

B.3.3 Hart State

The model state of a single hart comprises:

- *hart_id*, a unique identifier of the hart;
- *initial_register_state*, the initial register value for each register;
- *initial_fetch_address*, the initial instruction fetch address;
- *instruction_tree*, a tree of the instruction instances that have been fetched (and not discarded), in program order.

B.3.4 Shared Memory State

The model state of the shared memory comprises a list of memory store operations, in the order they propagated to the shared memory.

When a store operation is propagated to the shared memory it is simply added to the end of the list. When a load operation is satisfied from memory, for each byte of the load operation, the most recent corresponding store slice is returned.

For most purposes, it is simpler to think of the shared memory as an array, i.e., a map from memory locations to memory store operation slices, where each memory location is mapped to a one-byte slice of the most recent memory store operation to that location. However, this abstraction is not detailed enough to properly handle the `sc` *instruction. The RVWMO Atomicity Axiom allows store operations from the same hart as the* `sc` *to intervene between the store operation of the* `sc` *and the store operations the paired* `lr` *read from. To allow such store operations to intervene, and forbid others, the array abstraction must be extended to record more information. Here, we use a list as it is very simple, but a more efficient and scalable implementations should probably use something better.*

B.3.5 Transitions

Each of the paragraphs below describes a single kind of system transition. The description starts with a condition over the current system state. The transition can be taken in the current state only if the condition is satisfied. The condition is followed by an action that is applied to that state when the transition is taken, in order to generate the new system state.

Fetch instruction A possible program-order-successor of instruction instance i can be fetched from address *loc* if:

1. it has not already been fetched, i.e., none of the immediate successors of i in the hart's *instruction_tree* are from *loc*; and
2. if i's pseudocode has already wrote an address to *pc*, then *loc* must be that address, otherwise *loc* is:
   - for a conditional branch, the successor address and the branch target address;
   - for a (direct) jump and link instruction (`jal`), the target address;
   - for an indirect jump instruction (`jalr`), any address; and
   - for any other instruction, $i.program\_loc + 4$.

Action: construct a freshly initialized instruction instance i' for the instruction in the program memory at *loc*, with state PLAIN(*isa_state*), computed from the instruction pseudocode, including the static information available from the pseudocode such as its *instruction_kind*, *src_regs*, and *dst_regs*, and add i' to the hart's *instruction_tree* as a successor of i.

The possible next fetch addresses (loc) are available immediately after fetching i and the model does not need to wait for the pseudocode to write to pc*; this allows out-of-order execution, and speculation past conditional branches and jumps. For most instructions these addresses are easily obtained from the instruction pseudocode. The only exception to that is the indirect jump instruction (*`jalr`*), where the address depends on the value held in a register. In principle the mathematical model should allow speculation to arbitrary addresses here. The exhaustive search in the* `rmem` *tool handles this by running the exhaustive search multiple times with a growing set of possible next fetch addresses for each indirect jump. The initial search uses empty sets, hence there is no fetch after indirect jump instruction until the pseudocode of the instruction writes to* pc*, and then we use that value for fetching the next instruction. Before starting the next iteration of exhaustive search, we collect for each indirect jump (grouped by code location) the set of values it wrote to* pc *in all the executions in the previous search iteration, and use that as possible next fetch addresses of the instruction. This process terminates when no new fetch addresses are detected.*

Initiate memory load operations An instruction instance i in state PLAIN(LOAD_MEM(*kind*, *address*, *size*, *load_continuation*)) can always initiate the corresponding memory load operations. Action:

1. Construct the appropriate memory load operations *mlos*:

- if *address* is aligned to *size* then *mlos* is a single memory load operation of *size* bytes from *address*;
- otherwise, *mlos* is a set of *size* memory load operations, each of one byte, from the addresses $address \ldots address + size - 1$.

2. set *mem_loads* of i to *mlos*; and
3. update the state of i to PENDING_MEM_LOADS(*load_continuation*).

In Section 6.1 it is said that misaligned memory accesses may be decomposed at any granularity. Here we decompose them to one-byte accesses as this granularity subsumes all others.

Satisfy memory load operation by forwarding from unpropagated stores For a non-AMO load instruction instance i in state PENDING_MEM_LOADS(*load_continuation*), and a memory load operation *mlo* in i.*mem_loads* that has unsatisfied slices, the memory load operation can be partially or entirely satisfied by forwarding from unpropagated memory store operations by store instruction instances that are program-order-before i if:

1. all program-order-previous `fence` instructions with `.sr` and `.pw` set are finished;
2. for every program-order-previous `fence` instruction, f, with `.sr` and `.pr` set, and `.pw` not set, if f is not finished then all load instructions that are program-order-before f are entirely satisfied;
3. for every program-order-previous `fence.tso` instruction, f, that is not finished, all load instructions that are program-order-before f are entirely satisfied;
4. if i is a load-acquire-RCsc, all program-order-previous store-releases-RCsc are finished;
5. if i is a load-acquire-release, all program-order-previous instructions are finished;
6. all non-finished program-order-previous load-acquire instructions are entirely satisfied; and
7. all program-order-previous store-acquire-release instructions are finished;

Let *msoss* be the set of all unpropagated memory store operation slices from non-`sc` store instruction instances that are program-order-before i and have already calculated the value to be stored, that overlap with the unsatisfied slices of *mlo*, and which are not superseded by intervening stores operations or store operations that are read from by an intervening loads. The last condition requires, for each memory store operation slice *msos* in *msoss* from instruction i':

- that there is no store instruction program-order-between i and i' with a memory store operation overlapping *msos*; and
- that there is no load instruction program-order-between i and i' that was satisfied from an overlapping memory store operation slice from a different hart.

Action:

1. update i.*mem_loads* to indicate that *mlo* was satisfied by *msoss*; and

2. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' that is a program-order-successor of i, and every memory load operation mlo' of i' that was satisfied from $msoss'$, if there exists a memory store operation slice $msos'$ in $msoss'$, and an overlapping memory store operation slice from a different memory store operation in $msoss$, and $msos'$ is not from an instruction that is a program-order-successor of i, restart i' and its *restart-dependents*.

Where, the *restart-dependents* of instruction j are:

- program-order-successors of j that have data-flow dependency on a register write of j;
- program-order-successors of j that have a memory load operation that reads from a memory store operation of j (by forwarding);
- if j is a load-acquire, all the program-order-successors of j;
- if j is a load, for every `fence`, f, with `.sr` and `.pr` set, and `.pw` not set, that is program-order-successors of j, all the load instructions that are program-order-successors of f;
- if j is a load, for every `fence.tso`, f, that is program-order-successors of j, all the load instructions that are program-order-successors of f; and
- (recursively) all the restart-dependents of all the instruction instances above.

Forwarding memory store operations to a memory load might satisfy only some slices of the load, leaving other slices unsatisfied.

A program-order-previous store operation that was not available when taking the transition above might make msoss provisionally unsound (violating coherence) when it becomes available. That store will prevent the load from being finished (see Finish instruction), and will cause it to restart when that store operation is propagated (see Propagate store operation).

A consequence of the transition condition above is that store-release-RCsc memory store operations cannot be forwarded to load-acquire-RCsc instructions: msoss does not include memory store operations from finished stores (as those must be propagated memory store operations), and the condition above requires all program-order-previous store-releases-RCsc to be finished when the load is acquire-RCsc.

Satisfy memory load operation from memory For an instruction instance i of a non-AMO load instruction or an AMO instruction in the context of the "Satisfy, commit and propagate operations of an AMO" transition, any memory load operation mlo in $i.mem\_loads$ that has unsatisfied slices, can be satisfied from memory if all the conditions of Satisfy memory load operation by forwarding from unpropagated stores are satisfied. Action: let $msoss$ be the memory store operation slices from memory covering the unsatisfied slices of mlo, and apply the action of Satisfy memory load operation by forwarding from unpropagated stores.

Note that Satisfy memory load operation by forwarding from unpropagated stores might leave some slices of the memory load operation unsatisfied, those will have to be satisfied by taking the transition again, or taking Satisfy memory load operation from memory. Satisfy memory load operation from memory, on the other hand, will always satisfy all the unsatisfied slices of the memory load operation.

Complete load operations A load instruction instance i in state PENDING_MEM_LOADS(*load_continuation*) can be completed (not to be confused with finished) if all the memory load operations $i.mem\_loads$ are entirely satisfied (i.e. there are no unsatisfied

slices). Action: update the state of i to PLAIN(*load_continuation(mem_value)*), where *mem_value* is assembled from all the memory store operation slices that satisfied $i.mem\_loads$.

Early `sc` fail An `sc` instruction instance i in state PLAIN(EARLY_SC_FAIL(*res_continuation*)) can always be made to fail. Action: update the state of i to PLAIN(*res_continuation(false)*).

Paired `sc` An `sc` instruction instance i in state PLAIN(EARLY_SC_FAIL(*res_continuation*)) can continue its (potentially successful) execution if i is paired with an `lr`. Action: update the state of i to PLAIN(*res_continuation(true)*).

Initiate memory store operation footprints An instruction instance i in state PLAIN(STORE_EA(*kind*, *address*, *size*, *next_state*)) can always announce its pending memory store operation footprint. Action:

1. construct the appropriate memory store operations *msos* (without the store value):
   - if *address* is aligned to *size* then *msos* is a single memory store operation of *size* bytes to *address*;
   - otherwise, *msos* is a set of *size* memory store operations, each of one-byte size, to the addresses $address \ldots address + size - 1$.
2. set $i.mem\_stores$ to *msos*; and
3. update the state of i to PLAIN(*next_state*).

Note that after taking the transition above the memory store operations do not yet have their values. The importance of splitting this transition from the transition below is that it allows other program-order-successor store instructions to observe the memory footprint of this instruction, and if they don't overlap, propagate out of order as early as possible (i.e. before the data register value becomes available).

Instantiate memory store operation values An instruction instance i in state PLAIN(STORE_MEMV(*mem_value*, *store_continuation*)) can always instantiate the values of the memory store operations $i.mem\_stores$. Action:

1. split *mem_value* between the memory store operations $i.mem\_stores$; and
2. update the state of i to PENDING_MEM_STORES(*store_continuation*).

Commit store instruction An uncommitted instruction instance i of a non-`sc` store instruction or an `sc` instruction in the context of the "Commit and propagate store operation of an `sc`" transition, in state PENDING_MEM_STORES(*store_continuation*), can be committed (not to be confused with propagated) if:

1. i has fully determined data;
2. all program-order-previous conditional branch and indirect jump instructions are finished;
3. all program-order-previous `fence` instructions with `.sw` set are finished;
4. all program-order-previous `fence.tso` instructions are finished;
5. all program-order-previous load-acquire instructions are finished;
6. all program-order-previous store-acquire-release instructions are finished;
7. if i is a store-release, all program-order-previous instructions are finished;
8. all program-order-previous memory access instructions have a fully determined memory footprint;
9. all program-order-previous store instructions, except for `sc` that failed, have initiated and so have non-empty *mem_stores*; and
10. all program-order-previous load instructions have initiated and so have non-empty *mem_loads*.

Action: record that i is committed.

Notice that if condition 8 is satisfied the conditions 9 and 10 are also satisfied, or will be satisfied after taking some eager transitions. Hence, requiring them does not strengthen the model. By requiring them, we guarantee that previous memory access instructions have taken enough transitions to make their memory operations visible for the condition check of Propagate store operation, which is the next transition the instruction will take, making that condition simpler.

Propagate store operation For a committed instruction instance i in state PENDING_MEM_STORES(*store_continuation*), and an unpropagated memory store operation *mso* in i.*mem_stores*, *mso* can be propagated if:

1. all memory store operations of program-order-previous store instructions that overlap with *mso* have already propagated;
2. all memory load operations of program-order-previous load instructions that overlap with *mso* have already been satisfied, and (the load instructions) are *non-restartable* (see definition below); and
3. all memory load operations that were satisfied by forwarding *mso* are entirely satisfied.

Where a non-finished instruction instance j is *non-restartable* if:

1. there does not exist a store instruction s and an unpropagated memory store operation *mso* of s such that applying the action of the "Propagate store operation" transition to *mso* will result in the restart of j; and

2. there does not exist a non-finished load instruction l and a memory load operation mlo of l such that applying the action of the "Satisfy memory load operation by forwarding from unpropagated stores"/"Satisfy memory load operation from memory" transition (even if mlo is already satisfied) to mlo will result in the restart of j.

Action:

1. update the shared memory state with mso;
2. update $i.mem\_stores$ to indicate that mso was propagated; and
3. restart any speculative instructions which have violated coherence as a result of this, i.e., for every non-finished instruction i' program-order-after i and every memory load operation mlo' of i' that was satisfied from $msoss'$, if there exists a memory store operation slice $msos'$ in $msoss'$ that overlaps with mso and is not from mso, and $msos'$ is not from a program-order-successor of i, restart i' and its *restart-dependents* (see Satisfy memory load operation by forwarding from unpropagated stores).

Commit and propagate store operation of an `sc` An uncommitted `sc` instruction instance i, from hart h, in state PENDING_MEM_STORES(*store_continuation*), with a paired `lr` i' that has been satisfied by some store slices $msoss$, can be committed and propagated at the same time if:

1. i' is finished;
2. every memory store operation that has been forwarded to i' is propagated;
3. the conditions of Commit store instruction is satisfied;
4. the conditions of Propagate store operation is satisfied (notice that an `sc` instruction can only have one memory store operation); and
5. for every store slice $msos$ from $msoss$, $msos$ has not been overwritten, in the shared memory, by a store that is from a hart that is not h, at any point since $msos$ was propagated to memory.

Action:

1. apply the actions of Commit store instruction; and
2. apply the action of Propagate store operation.

Late `sc` fail An `sc` instruction instance i in state PENDING_MEM_STORES(*store_continuation*), that has not propagated its memory store operation, can always be made to fail. Action:

1. clear $i.mem\_stores$; and
2. update the state of i to PLAIN(*store_continuation(false)*).

For efficiency, the `rmem` *tool allows this transition only when it is not possible to take the Commit and propagate store operation of an* `sc` *transition. This does not affect the set of allowed final states, but when explored interactively, if the* `sc` *should fail one should use the Early* `sc` *fail transition instead of waiting for this transition.*

Complete store operations A store instruction instance i in state PENDING_MEM_STORES(*store_continuation*), for which all the memory store operations in i.*mem_stores* have been propagated, can always be completed (not to be confused with finished). Action: update the state of i to PLAIN(*store_continuation(true)*).

Satisfy, commit and propagate operations of an AMO An AMO instruction instance i in state PENDING_MEM_LOADS(*load_continuation*) can perform its memory access if it is possible to perform the following sequence of transitions with no intervening transitions:

1. Satisfy memory load operation from memory
2. Complete load operations
3. Pseudocode internal step (zero or more times)
4. Instantiate memory store operation values
5. Commit store instruction
6. Propagate store operation
7. Complete store operations

Action: perform the above sequence of transitions, one after the other, with no intervening transitions.

Notice that program-order-previous stores cannot be forwarded to the load of an AMO. This is simply because the sequence of transitions above does not include the forwarding transition. But even if it did include it, the sequence will fail when trying to do the Propagate store operation transition, as this transition requires all program-order-previous store operations to overlapping memory footprints to be propagated, and forwarding requires the store operation to be unpropagated.

In addition, the store of an AMO cannot be forwarded to a program-order-successor load. Before taking the transition above, the store operation of the AMO does not have its value and therefore cannot be forwarded; after taking the transition above the store operation is propagated and therefore cannot be forwarded.

Commit fence A fence instruction instance i in state PLAIN(FENCE(*kind*, *next_state*)) can be committed if:

1. if i is a normal fence and it has `.pr` set, all program-order-previous load instructions are finished;

2. if i is a normal fence and it has `.pw` set, all program-order-previous store instructions are finished; and
3. if i is a `fence.tso`, all program-order-previous load and store instructions are finished.

Action:

1. record that i is committed; and
2. update the state of i to PLAIN(*next_state*).

Register read An instruction instance i in state PLAIN(READ_REG(*reg_name*, *read_cont*)) can do a register read of *reg_name* if every instruction instance that it needs to read from has already performed the expected *reg_name* register write.

Let *read_sources* include, for each bit of *reg_name*, the write to that bit by the most recent (in program order) instruction instance that can write to that bit, if any. If there is no such instruction, the source is the initial register value from *initial_register_state*. Let *reg_value* be the value assembled from *read_sources*. Action:

1. add *reg_name* to *i.reg_reads* with *read_sources* and *reg_value*; and
2. update the state of i to PLAIN(*read_cont(reg_value)*).

Register write An instruction instance i in state PLAIN(WRITE_REG(*reg_name*, *reg_value*, *next_state*)) can always do a *reg_name* register write. Action:

1. add *reg_name* to *i.reg_writes* with *deps* and *reg_value*; and
2. update the state of i to PLAIN(*next_state*).

where *deps* is a pair of the set of all *read_sources* from *i.reg_reads*, and a flag that is true iff i is a load instruction instance that has already been entirely satisfied.

Pseudocode internal step An instruction instance i in state PLAIN(INTERNAL(*next_state*)) can always do that pseudocode-internal step. Action: update the state of i to PLAIN(*next_state*).

Finish instruction A non-finished instruction instance i in state PLAIN(DONE) can be finished if:

1. if i is a load instruction:
   (a) all program-order-previous load-acquire instructions are finished;
   (b) all program-order-previous `fence` instructions with `.sr` set are finished;

(c) for every program-order-previous `fence.tso` instruction, f, that is not finished, all load instructions that are program-order-before f are finished; and

(d) it is guaranteed that the values read by the memory load operations of i will not cause coherence violations, i.e., for any program-order-previous instruction instance i', let *cfp* be the combined footprint of propagated memory store operations from store instructions program-order-between i and i', and *fixed memory store operations* that were forwarded to i from store instructions program-order-between i and i' including i', and let $\overline{cfp}$ be the complement of *cfp* in the memory footprint of i. If $\overline{cfp}$ is not empty:

i. i' has a fully determined memory footprint;

ii. i' has no unpropagated memory store operations that overlap with $\overline{cfp}$; and

iii. if i' is a load with a memory footprint that overlaps with $\overline{cfp}$, then all the memory load operations of i' that overlap with $\overline{cfp}$ are satisfied and i' is *non-restartable* (see the Propagate store operation transition for how to determined if an instruction is non-restartable).

Here, a memory store operation is called fixed if the store instruction has fully determined data.

2. i has a fully determined data; and

3. if i is not a fence, all program-order-previous conditional branch and indirect jump instructions are finished.

Action:

1. if i is a conditional branch or indirect jump instruction, discard any untaken paths of execution, i.e., remove all instruction instances that are not reachable by the branch/jump taken in *instruction_tree*; and

2. record the instruction as finished, i.e., set *finished* to *true*.

B.3.6 Limitations

- The model covers user-level RV64I and RV64A. In particular, it does not support the misaligned atomics extension "Zam" or the total store ordering extension "Ztso". It should be trivial to adapt the model to RV32I/A and to the G, Q and C extensions, but we have never tried it. This will involve, mostly, writing Sail code for the instructions, with minimal, if any, changes to the concurrency model.
- The model covers only normal memory accesses (it does not handle I/O accesses).
- The model does not cover TLB-related effects.
- The model assumes the instruction memory is fixed. In particular, the Fetch instruction transition does not generate memory load operations, and the shared memory is not involved in the transition. Instead, the model depends on an external oracle that provides an opcode when given a memory location.
- The model does not cover exceptions, traps and interrupts.

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] Heidi Pan, Benjamin Hindman, and Krste Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar '09)*, Berkeley, CA, March 2009.

[24] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with Lithe. In *31st Conference on Programming Language Design and Implementation*, Toronto, Canada, June 2010.

[25] David A. Patterson and Carlo H. Séquin. RISC I: A reduced instruction set VLSI computer. In *ISCA*, pages 443–458, 1981.

[26] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42 –50, August 1996.

[27] 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.

[28] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium-III processor. *IEEE Micro*, 20(4):47 –57, 2000.

[29] 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.

[30] 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.

[31] M. Tremblay, J.M. O'Connor, V. Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10 –20, August 1996.

[32] 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.

[33] 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.

[34] 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.

[35] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master's thesis, University of California, Berkeley, 2011.

[36] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, 2016.

[37] 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.

[38] 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.