Requirements List

of

The RISC-V Instruction Set Manual

Volume I: Unprivileged ISA

Document Version 20191213*

by

Mehmet Öner

Version 1.0

*"The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019.
https://riscv.org/technical/specifications/

About

The Document

In systems engineering approach, before doing anything with the design of the system under consideration, *requirements analysis* must be completed as one of the first tasks (if not the very first). Beginning with an itemized, atomic, classified and well defined list of requirements is essential. Because, following activities at various stages of development (like design coverage analysis, testing, verification, validation …) depend on the requirement specifications stated at the beginning.

RISC-V International provides the ISA (Instruction Set Architecture) and non-ISA requirement specifications for the RISC-V architecture (https://riscv.org/technical/specifications/). These documents in general are good written technical plain text documents. However, they lack some aspects of good requirement specification practices:

- Requirements are in free text form and not itemized: Itemized list of requirements enables requirement coverage in design, test, verification and validation phases.
- Text includes comments and information statements along with requirements: Statements must be clearly labeled and categorized.
- Some statements include more than one specifications: Each specification need to be isolated.
- Some specifications such as instruction definitions are distributed throughout the text: The distributed content need to be put together to have a complete the specification.

The aim of this document is providing an edited list of requirements for “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213”, Editors Andrew Waterman and Krste Asanović, RISC-V Foundation, December 2019. https://riscv.org/technical/specifications/. Released under a Creative Commons Attribution 4.0 International License. https://creativecommons.org/licenses/by/4.0/.

In particular:

- Statements were itemized and given an ID number.
- Each itemized statement was referenced to the original document to provide traceability
- Itemized statements were categorized
- Complex statements were broken into simpler atomic requirement statements when needed.
- Distributed requirement information was put together to form complete specifications.

Special attention was given to preserve original statements, even when dividing complex statements into simpler atomic statements. But occasionally, some statements were re-written as to form a formal requirement statement.

Following table lists which chapters are included or excluded from this document and why.

Chapter	**Status**	**Explanation**
Chapter 1 Introduction	Included	This chapter includes introductory materials and some requirements
Chapter 2 RV32I Base Integer Instruction Set	Included	RV32I is ratified
Chapter 3 “Zifencei” Instruction-Fetch Fence	Included	Zifencei extension is ratified
Chapter 4 RV32E Base Integer Instruction Set	Included	RV32E is not ratified, but the current state of the extension is given clearly.

Chapter	**Status**	**Explanation**
Chapter 5 RV64I Base Integer Instruction Set	Included	RV64I is ratified
Chapter 6 RV128I Base Integer Instruction Set	Included	RV128I is not ratified. However except some new instructions that would came with RV128I, its current status is given. Yet RV128I is referenced in many other instructions.
Chapter 7 “M” Standard Extension for Integer Multiplication and Division,	Included	M extension is ratified
Chapter 8 “A” Standard Extension for Atomic Instructions	Included	A extension is ratified
Chapter 9 “Zicsr”, Control and Status Register (CSR) Instructions,	Included	Zicsr extension is ratified
Chapter 10 Counters	Included	Counters extension is ratified
Chapter 11 “F” Standard Extension for Single-Precision Floating-Point	Included	F extension is ratified
Chapter 12 “D” Standard Extension for Double-Precision Floating-Point	Included	D extension is ratified
Chapter 13 “Q” Standard Extension for Quad-Precision Floating-Point	Included	Q extension is ratified
Chapter 14 RVWMO Memory Consistency Model	Included	RVWMO model is ratified
Chapter 15 “L” Standard Extension for Decimal Floating-Point	Excluded	L extension is not ratified
Chapter 16 “C” Standard Extension for Compressed Instructions	Included	C extension is ratified
Chapter 17 “B” Standard Extension for Bit Manipulation	Excluded	B extension is not ratified
Chapter 18 “J” Standard Extension for Dynamically Translated Languages	Excluded	J extension is not ratified
Chapter 19 “T” Standard Extension for Transactional Memory,	Excluded	T extension is not ratified
Chapter 20 “P” Standard Extension for Packed-SIMD Instructions	Excluded	P extension is not ratified
Chapter 21 “V” Standard Extension for Vector Operations	Excluded	V extension is not ratified
Chapter 22 “Zam” Standard Extension for Misaligned Atomics	Included	Zam extension is ratified
Chapter 23 “Ztso” Standard Extension for Total Store Ordering	Included	Ztso extension is ratified
Chapter 24 RV32/64G Instruction Set Listings	Included	Information about meta G extension
Chapter 25 RISC-V Assembly Programmer’s Handbook	Excluded	This chapter is yet a place holder for assembly programers manual.
Chapter 26 Extending RISC-V	Excluded	This chapter is about extending RISC-V.
Chapter 27 ISA Extension Naming Conventions	Excluded	This chapter is about naming conventions of RISC-V extensions.
Chapter 28 History and Acknowledgments	Excluded	This chapter provides history of RISC-V.

Chapter	Status	Explanation
Appendix A RVWMO Explanatory Material	Excluded	RVWMO memory model is explained.
Appendix B Formal Memory Model Specifications	Excluded	Formal analysis of RVWMO memory model.

The Editor (or the systems engineer)

After 34 years of my career, I retired from my regular job in 2023. Now, I do part time consulting services to interested parties, while I do work on projects that interest me more than a regular work.

In my career I dealt with very diverse fields of engineering: Academics, C/C++ desktop programming, embedded systems, analog circuit design, DSP algorithms, VLSI/FPGA design, underwater acoustics are to name few.

I've always enjoyed designing controllers and processors with generic HDL for VLSI or FPGA. So, as my personal project to work on, I decided to design RISC-V cores with different capabilities.

Having some defense sector background, I find systems engineering approach very useful. After reading RISC-V specification documents, I decided to take the initiative and edit the documents into customer requirements list format which is a tough, tedious and time consuming work.

In case someone else could find these documents useful, I share them on github:
https://github.com/vizionerco/RISC-V

Best regards,

Mehmet Öner, Ph.D.

www.linkedin.com/in/mehmet-oner-00453733/

Table of Contents

Definitions

Table 1: Requirement Types

TYPE	NAME	EXPLANATION
H	Heading	Headings in the original document. Headings are included to provide context to the subsequent requirements
I	Information	These are statements that explain some aspects of the subject, but actually do not specify any requirement. For these types, the statement(s) in the text is given as is.
C	Comment	These statements are original comment statements in the RISC_V documentation which are explained as: “Commentary on our design decisions is formatted as in this paragraph. This non-normative text can be skipped if the reader is only interested in the specification itself.” For these types, the statement(s) in the text is given as is.
R	Requirement	These are statements that specify a specific need to be fulfilled. For these types, the statement(s) in the text is given as is where possible. In some cases, information is collected from different tables and figures to form a complete specification. In some cases, context is added in parenthesis in gray color to make the requirement self explanatory. In some cases, the statements are broken into single requirement statements.
O	Optional Requirement	These are statements that specify a property that is not mandatory to implement. However if it is chosen to fulfill, it should obey this requirement. Inclusion of the text is the same as R/Requirement type.
T	Tentative Requirement	These are requirements but are not frozen yet by the RISC-V committee. Inclusion of the text is the same as R/Requirement type.

Table 2: Abbreviations & Definitions

SHORT	MEANING
ABI	Application binary interface
Accelerator	either a non-programmable fixed-function unit or a core that can operate autonomously but is specialized for certain tasks.
AEE	Application Execution Environment
Bare Metal EEI	Hardware platforms where harts are directly implemented by physical processor threads and instructions have full access to the physical address space. The hardware platform defines an execution environment that begins at power-on reset.
Contained Trap	The trap is visible to, and handled by, software running inside the execution environment. For example, in an EEI providing both supervisor and user mode on harts, an ECALL by a user-mode hart will generally result in a transfer of control to a supervisor mode handler running on the same hart. Similarly, in the same environment, when a hart is interrupted, an interrupt handler will be run in supervisor mode on the hart.
Coprocessor	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.
Core	A component is termed a core if it contains an independent instruction fetch unit
CSR	Control and Status Registers
EEI	Execution environment interface

SHORT	MEANING
Emulator EEI	Emulates RISC-V harts on an underlying operating system, and which can provide either a user-level or a supervisor-level execution environment.
Exception	We use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.
Fatal Trap	The trap represents a fatal failure and causes the execution environment to terminate execution. Examples include failing a virtual-memory page-protection check or allowing a watchdog timer to expire. Each EEI should define how execution is terminated and reported to an external environment.
FLEN	We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA
Foundation	RISC-V Foundation
Hart	Hardware Tread. From the perspective of software running in a given execution environment, a hart is a resource that autonomously fetches and executes RISC-V instructions within that execution environment. In this respect, a hart behaves like a hardware thread resource even if time-multiplexed onto real hardware by the execution environment.
HBI	Hypervisor binary interface
HEE	Hypervisor execution environment
Hypervisor EEI	Provides multiple supervisor-level execution environments for guest operating systems.
IALIGN	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.
ILEN	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.
Interrupt	We use the term *interrupt* to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.
Invisible Trap	The trap is handled transparently by the execution environment and execution resumes normally after the trap is handled. Examples include emulating missing instructions, handling non-resident page faults in a demand-paged virtual-memory system, or handling device interrupts for a different job in a multiprogrammed machine. In these cases, the software running inside the execution environment is not aware of the trap (we ignore timing effects in these definitions).
ISA	Instruction Set Architecture
Memory Word	A *word* of memory is defined as 32 bits (4 bytes). Correspondingly, a *halfword* is 16 bits (2 bytes), a *doubleword* is 64 bits (8 bytes), and a *quadword* is 128 bits (16 bytes).
Nonstandard ISA Extension	An extension that is not defined by the RISC-V Foundation.
Operating System EEI	Provides multiple user-level execution environments by multiplexing user-level harts onto available physical processor threads and by controlling access to memory via virtual memory.
OS	Operating System
PMA	Physical memory attributes
PTE	Page Table Entry

SHORT	MEANING
RC	Release Consistency (memory consistency model)
RCpc	Release consistency with processor-consistent synchronization operations
RCsc	Release Consistency with sequentially-consistent synchronization operations
Requested Trap	The trap is a synchronous exception that is an explicit call to the execution environment requesting an action on behalf of software inside the execution environment. An example is a system call. In this case, execution may or may not resume on the hart after the requested action is taken by the execution environment. For example, a system call could remove the hart or cause an orderly termination of the entire execution environment.
RV32GC	RV32 ISA with G pseudo extension and C extension
RV32I	RISC-V 32-bit Base Integer ISA
RV64GC	RV64 ISA with G pseudo extension and C extension
RV64I	RISC-V 64-bit Base Integer ISA
SBI	RISC-V supervisor binary interface
SEE	Supervisor Execution Environment
Trap	We use the term *trap* to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.
TLB	Translation lookaside buffer
UNSPECIFIED	The term UNSPECIFIED refers to a behavior or value that is intentionally unconstrained. The definition of these behaviors or values is open to extensions, platform standards, or implementations. Extensions, platform standards, or implementation documentation may provide normative content to further constrain cases that the base architecture defines as UNSPECIFIED. Like the base architecture, extensions should fully describe allowable behavior and values and use the term UNSPECIFIED for cases that are intentionally unconstrained. These cases may be constrained or defined by other extensions, platform standards, or implementations.
XLEN	Refers to the width of an integer register in bits of an ISA

CHAPTER 1 Introduction

ID	REFERENCE	TYPE	DEFINITION
RVU.1.1	1.0 (p.1)	H	Introduction
RVU.1.2	1.0 (p.1)	I	RISC-V (pronounced “risk-five”) is a new instruction-set architecture (ISA) that was originally designed to support computer architecture research and education, but which we now hope will also become a standard free and open architecture for industry implementations. Our goals in defining RISC-V include: • A completely open ISA that is freely available to academia and industry. • A real ISA suitable for direct native hardware implementation, not just simulation or binary translation. • An ISA that avoids “over-architecting” for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these. • An ISA separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support generalpurpose software development. • Support for the revised 2008 IEEE-754 floating-point standard†. • 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.
RVU.1.3	1.0 (p.1, p.2)	C	The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I, RISC-II, SOAR, and SPUR 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.
RVU.1.4	1.0 (p.2)	I	A fully virtualizable ISA to ease hypervisor development.The RISC-V ISA is defined avoiding implementation details as much as possible (although commentary is included on implementation-driven decisions) and should be read as the software-visible interface to a wide variety of implementations rather than as the design of a particular hardware artifact.
RVU.1.5	1.0 (p.2)	I	The RISC-V manual is structured in two volumes. This volume covers the design of the base unprivileged instructions, including optional unprivileged ISA extensions. Unprivileged instructions are those that are generally usable in all privilege modes in all privileged architectures, though behavior might vary depending on privilege mode and privilege architecture.

†ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008

ID	REFERENCE	TYPE	DEFINITION
RVU.1.6	1.0 (p.2)	I	The second volume provides the design of the first (“classic”) privileged architecture.
RVU.1.7	1.0 (p.2)	I	The manuals use IEC 80000-13:2008 conventions, ...
RVU.1.8	1.0 (p.2)	R	… (IEC 80000-13:2008 conventions,) with a byte of 8 bits.
RVU.1.9	1.0 (p.2)	C	In the unprivileged ISA design, we tried to remove any dependence on particular microarchitectural features, such as cache line size, or on privileged architecture details, such as page translation. This is both for simplicity and to allow maximum flexibility for alternative microarchitectures or alternative privileged architectures.
RVU.1.10	1.1 (p.2)	H	RISC-V Hardware Platform Terminology
RVU.1.11	1.1 (p.2)	I	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.
RVU.1.12	1.1 (p.2)	I	A component is termed a *core* if it contains an independent instruction fetch unit.
RVU.1.13	1.1 (p.2)	I	A RISC-V compatible *core* might support multiple RISC-V-compatible hardware threads, or *harts*, through multithreading.
RVU.1.14	1.1 (p.2)	I	A RISC-V core might have additional specialized instruction-set extensions or an added *coprocessor*
RVU.1.15	1.1 (p.2)	I	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.
RVU.1.16	1.1 (p.2)	I	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.
RVU.1.17	1.1 (p.2)	I	In RISC-V systems, we expect many programmable accelerators will be RISC-V-based cores with specialized instruction-set extensions and/or customized coprocessors.
RVU.1.18	1.1 (p.2)	I	An important class of RISC-V accelerators are I/O accelerators, which offload I/O processing tasks from the main application cores.
RVU.1.19	1.1 (p.2)	I	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.
RVU.1.20	1.2 (p.3)	H	RISC-V Software Execution Environments and Harts

ID	REFERENCE	TYPE	DEFINITION
RVU.1.21	1.2 (p.3)	I	The behavior of a RISC-V program depends on the execution environment in which it runs. A RISC-V execution environment interface (EEI) defines the initial state of the program, the number and type of harts in the environment including the privilege modes supported by the harts, the accessibility and attributes of memory and I/O regions, the behavior of all legal instructions executed on each hart (i.e., the ISA is one component of the EEI), and the handling of any interrupts or exceptions raised during execution including environment calls. Examples of EEIs include the Linux application binary interface (ABI), or the RISC-V supervisor binary interface (SBI).
RVU.1.22	1.2 (p.3)	I	The implementation of a RISC-V execution environment can be pure hardware, pure software, or a combination of hardware and software. For example, opcode traps and software emulation can be used to implement functionality not provided in hardware. Examples of execution environment implementations include: • “Bare metal” hardware platforms where harts are directly implemented by physical processor threads and instructions have full access to the physical address space. The hardware platform defines an execution environment that begins at power-on reset. • RISC-V operating systems that provide multiple user-level execution environments by multiplexing user-level harts onto available physical processor threads and by controlling access to memory via virtual memory. • RISC-V hypervisors that provide multiple supervisor-level execution environments for guest operating systems. • RISC-V emulators, such as Spike, QEMU or rv8, which emulate RISC-V harts on an underlying x86 system, and which can provide either a user-level or a supervisor-level execution environment.
RVU.1.23	1.2 (p.3)	C	A bare hardware platform can be considered to define an EEI, where the accessible harts, memory, and other devices populate the environment, and the initial state is that at power-on reset. Generally, most software is designed to use a more abstract interface to the hardware, as more abstract EEIs provide greater portability across different hardware platforms. Often EEIs are layered on top of one another, where one higher-level EEI uses another lower-level EEI.
RVU.1.24	1.2 (p.3)	I	From the perspective of software running in a given execution environment, a hart is a resource that autonomously fetches and executes RISC-V instructions within that execution environment. In this respect, a hart behaves like a hardware thread resource even if time-multiplexed onto real hardware by the execution environment. Some EEIs support the creation and destruction of additional harts, for example, via environment calls to fork new harts.
RVU.1.25	1.2 (p.3-4)	I	The execution environment is responsible for ensuring the eventual forward progress of each of its harts. The following events constitute forward progress: • The retirement of an instruction. • A trap, as defined in Section 1.6. • Any other event defined by an extension to constitute forward progress.
RVU.1.26	1.2 (p.3)	I	The responsibility of the execution environment is suspended for a given hart, while the hart is exercising a mechanism that explicitly waits for an event, such as the wait-for-interrupt instruction defined in Volume II of this specification; and that responsibility ends if the hart is terminated.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.27	1.2 (p.4)	C	The term hart was introduced in the work on Lithe‡ [13, 14] to provide a term to represent an abstract execution resource as opposed to a software thread programming abstraction.
RVU.1.28	1.2 (p.4)	C	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.
RVU.1.29	1.2 (p.4)	C	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.
RVU.1.30	1.3 (p.4)	H	RISC-V ISA Overview
RVU.1.31	1.3 (p.4)	I	A RISC-V ISA is defined as a base integer ISA plus optional extensions to the base ISA.
RVU.1.32	1.3 (p.4)	I	The base integer ISAs are very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. A base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional privileged operations), and so provides a convenient ISA and software toolchain “skeleton” around which more customized processor ISAs can be built.
RVU.1.33	1.3 (p.4)	I	Although it is convenient to speak of the RISC-V ISA, RISC-V is actually a family of related ISAs, of which there are currently four base ISAs.
RVU.1.34	1.3 (p.4)	I	Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the address space and by the number of integer registers.
RVU.1.35	1.3 (p.4)	I	There are two primary base integer variants, RV32I and RV64I, described in Chapters 2 and 5, which provide 32-bit or 64-bit address spaces respectively.
RVU.1.36	1.3 (p.4)	I	We use the term XLEN to refer to the width of an integer register in bits (either 32 or 64).
RVU.1.37	1.3 (p.4)	I	Chapter 4 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers, and which has half the number of integer registers.
RVU.1.38	1.3 (p.4)	I	Chapter 6 sketches a future RV128I variant of the base integer instruction set supporting a flat 128-bit address space (XLEN=128).
RVU.1.39	1.3 (p.4)	R	The base integer instruction sets use a two’s-complement representation for signed integer values.

‡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.
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.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.40	1.3 (p.4)	C	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.
RVU.1.41	1.3 (p.4)	C	The four base ISAs in RISC-V are treated as distinct base ISAs. A common question is why is there not a single ISA, and in particular, why is RV32I not a strict subset of RV64I? Some earlier ISA designs (SPARC, MIPS) adopted a strict superset policy when increasing address space size to support running existing 32-bit binaries on new 64-bit hardware.
RVU.1.42	1.3 (p.5)	C	The main advantage of explicitly separating base ISAs is that each base ISA can be optimized for its needs without requiring to support all the operations needed for other base ISAs. For example, RV64I can omit instructions and CSRs that are only needed to cope with the narrower registers in RV32I. The RV32I variants can use encoding space otherwise reserved for instructions only required by wider address-space variants.
RVU.1.43	1.3 (p.5)	C	The main disadvantage of not treating the design as a single ISA is that it complicates the hardware needed to emulate one base ISA on another (e.g., RV32I on RV64I). However, differences in addressing and illegal instruction traps generally mean some mode switch would be required in hardware in any case even with full superset instruction encodings, and the different RISC-V base ISAs are similar enough that supporting multiple versions is relatively low cost. Although some have proposed that the strict superset design would allow legacy 32-bit libraries to be linked with 64-bit code, this is impractical in practice, even with compatible encodings, due to the differences in software calling conventions and system-call interfaces.
RVU.1.44	1.3 (p.5)	C	The RISC-V privileged architecture provides fields in misa to control the unprivileged ISA at each level to support emulating different base ISAs on the same hardware. We note that newer SPARC and MIPS ISA revisions have deprecated support for running 32-bit code unchanged on 64-bit systems.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.45	1.3 (p.5)	C	A related question is why there is a different encoding for 32-bit adds in RV32I (ADD) and RV64I (ADDW)? The ADDW opcode could be used for 32-bit adds in RV32I and ADDD for 64-bit adds in RV64I, instead of the existing design which uses the same opcode ADD for 32-bit adds in RV32I and 64-bit adds in RV64I with a different opcode ADDW for 32-bit adds in RV64I. This would also be more consistent with the use of the same LW opcode for 32-bit load in both RV32I and RV64I. The very first versions of RISC-V ISA did have a variant of this alternate design, but the RISC-V design was changed to the current choice in January 2011. Our focus was on supporting 32-bit integers in the 64-bit ISA not on providing compatibility with the 32-bit ISA, and the motivation was to remove the asymmetry that arose from having not all opcodes in RV32I have a *W suffix (e.g., ADDW, but AND not ANDW). In hindsight, this was perhaps not well-justified and a consequence of designing both ISAs at the same time as opposed to adding one later to sit on top of another, and also from a belief we had to fold platform requirements into the ISA spec which would imply that all the RV32I instructions would have been required in RV64I. It is too late to change the encoding now, but this is also of little practical consequence for the reasons stated above.
RVU.1.46	1.3 (p.5)	C	It has been noted we could enable the *W variants as an extension to RV32I systems to provide a common encoding across RV64I and a future RV32 variant.
RVU.1.47	1.3 (p.5)	I	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 RISCV instruction-set encoding space (and related encoding spaces such as the CSRs) into three disjoint categories: *standard*, *reserved*, and *custom*.
RVU.1.48	1.3 (p.5)	I	Standard encodings are defined by the Foundation, and shall not conflict with other standard extensions for the same base ISA.
RVU.1.49	1.3 (p.5)	I	Reserved encodings are currently not defined but are saved for future standard extensions.
RVU.1.50	1.3 (p.5)	I	We use the term *non-standard* to describe an extension that is not defined by the Foundation.
RVU.1.51	1.3 (p.5)	I	Custom encodings shall never be used for standard extensions and are made available for vendor-specific non-standard extensions.
RVU.1.52	1.3 (p.5)	I	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).
RVU.1.53	1.3 (p.5)	I	Instruction-set extensions are generally shared but may provide slightly different functionality depending on the base ISA. Chapter 26 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 27.
RVU.1.54	1.3 (p.5)	I	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.
RVU.1.55	1.3 (p.6)	I	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.56	1.3 (p.6)	I	The standard integer multiplication and division extension is named “M”, and adds instructions to multiply and divide values held in the integer registers.
RVU.1.57	1.3 (p.6)	I	The standard atomic instruction extension, denoted by “A”, adds instructions that atomically read, modify, and write memory for inter-processor synchronization.
RVU.1.58	1.3 (p.6)	I	The standard single-precision floating-point extension, denoted by “F”, adds floating-point registers, single-precision computational instructions, and single-precision loads and stores.
RVU.1.59	1.3 (p.6)	I	The standard double-precision floating-point extension, denoted by “D”, expands the floating-point registers, and adds double-precision computational instructions, loads, and stores.
RVU.1.60	1.3 (p.6)	I	The standard “C” compressed instruction extension provides narrower 16-bit forms of common instructions.
RVU.1.61	1.3 (p.6)	I	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.
RVU.1.62	1.3 (p.6)	I	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.
RVU.1.63	1.4 (p.6)	H	Memory
RVU.1.64	1.4 (p.6)	R	A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes for all memory accesses.
RVU.1.65	1.3 (p.6)	I	A *word* of memory is defined as 32 bits (4 bytes).Correspondingly, a *halfword* is 16 bits (2 bytes), a *doubleword* is 64 bits (8 bytes), and a *quadword* is 128 bits (16 bytes).
RVU.1.66	1.4 (p.6)	R	The memory address space is circular, so that the byte at address $2^{XLEN}-1$ is adjacent to the byte at address zero. Accordingly, memory address computations done by the hardware ignore overflow and instead wrap around modulo 2^{XLEN}
RVU.1.67	1.4 (p.6)	I	The execution environment determines the mapping of hardware resources into a hart’s address space. Different address ranges of a hart’s address space may (1) be vacant, or (2) contain *main memory*, or (3) contain one or more *I/O devices*.
RVU.1.68	1.4 (p.6)	O	Reads and writes of I/O devices may have visible side effects, ...
RVU.1.69	1.4 (p.6)	R	..., but accesses to main memory cannot (have visible side effects).
RVU.1.70	1.4 (p.6)	R	Although it is possible for the execution environment to call everything in a hart’s address space an I/O device, it is usually expected that some portion will be specified as main memory.
RVU.1.71	1.4 (p.6)	I	When a RISC-V platform has multiple harts, the address spaces of any two harts may be entirely the same, or entirely different, or may be partly different but sharing some subset of resources, mapped into the same or different address ranges.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.72	1.4 (p.6)	C	For a purely “bare metal” environment, all harts may see an identical address space, accessed entirely by physical addresses. However, when the execution environment includes an operating system employing address translation, it is common for each hart to be given a virtual address space that is largely or entirely its own.
RVU.1.73	1.4 (p.7)	I	Executing each RISC-V machine instruction entails one or more memory accesses, subdivided into *implicit* and *explicit* accesses. For each instruction executed, an *implicit* memory read (instruction fetch) is done to obtain the encoded instruction to execute. Many RISC-V instructions perform no further memory accesses beyond instruction fetch. Specific load and store instructions perform an *explicit* read or write of memory at an address determined by the instruction. The execution environment may dictate that instruction execution performs other *implicit* memory accesses (such as to implement address translation) beyond those documented for the unprivileged ISA.
RVU.1.74	1.4 (p.7)	I	The execution environment determines what portions of the non-vacant address space are accessible for each kind of memory access. For example, the set of locations that can be implicitly read for instruction fetch may or may not have any overlap with the set of locations that can be explicitly read by a load instruction; and the set of locations that can be explicitly written by a store instruction may be only a subset of locations that can be read.
RVU.1.75	1.4 (p.7)	R	Ordinarily, if an instruction attempts to access memory at an inaccessible address, an exception is raised for the instruction.
RVU.1.76	1.4 (p.7)	R	Vacant locations in the address space are never accessible.
RVU.1.77	1.4 (p.7)	O	Except when specified otherwise, implicit reads that do not raise an exception and that have no side effects may occur arbitrarily early and speculatively, even before the machine could possibly prove that the read will be needed. For instance, a valid implementation could attempt to read all of main memory at the earliest opportunity, cache as many fetchable (executable) bytes as possible for later instruction fetches, and avoid reading main memory for instruction fetches ever again.
RVU.1.78	1.4 (p.7)	R	To ensure that certain implicit reads are ordered only after writes to the same memory locations, software must execute specific fence or cache-control instructions defined for this purpose (such as the FENCE.I instruction defined in Chapter 3).
RVU.1.79	1.4 (p.7)	I	The memory accesses (implicit or explicit) made by a hart may appear to occur in a different order as perceived by another hart or by any other agent that can access the same memory. This perceived reordering of memory accesses is always constrained, however, by the applicable memory consistency model.
RVU.1.80	1.4 (p.7)	O	The default memory consistency model for RISC-V is the RISC-V Weak Memory Ordering (RVWMO), defined in Chapter 14 and in appendices.
RVU.1.81	1.4 (p.7)	O	Optionally, an implementation may adopt the stronger model of Total Store Ordering, as defined in Chapter 23.
RVU.1.82	1.4 (p.7)	I	The execution environment may also add constraints that further limit the perceived reordering of memory accesses.
RVU.1.83	1.4 (p.7)	I	Since the RVWMO model is the weakest model allowed for any RISC-V implementation, software written for this model is compatible with the actual memory consistency rules of all RISC-V implementations.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.84	1.4 (p.7)	R	As with implicit reads, software must execute fence or cache-control instructions to ensure specific ordering of memory accesses beyond the requirements of the assumed memory consistency model and execution environment.
RVU.1.85	1.5 (p.7)	H	Base Instruction-Length Encoding
RVU.1.86	1.5 (p.7)	R	The base RISC-V ISA has fixed-length 32-bit instructions ...
RVU.1.87	1.5 (p.7)	R	...(instructions) that must be naturally aligned on 32-bit boundaries.
RVU.1.88	1.5 (p.7)	I	However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction *parcels* in length and parcels are naturally aligned on 16-bit boundaries. The standard compressed ISA extension described in Chapter 16 reduces code size by providing compressed 16-bit instructions and relaxes the alignment constraints to allow all instructions (16 bit and 32 bit) to be aligned on any 16-bit boundary to improve code density.
RVU.1.89	1.5 (p.8)	I	We use the term IALIGN (measured in bits) to refer to the instruction-address alignment constraint the implementation enforces.
RVU.1.90	1.5 (p.8)	R	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.
RVU.1.91	1.5 (p.8)	I	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.
RVU.1.92	1.5 (p.8)	R	For implementations supporting only a base instruction set, ILEN is 32 bits. Implementations supporting longer instructions have larger values of ILEN.
RVU.1.93	1.5 (p.8) Figure 1.1	R	The optional compressed 16-bit instruction-set extensions have their lowest two bits (b_1b_0) equal to ‘00’, ‘01’, or ‘10’.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	b_1	b_0

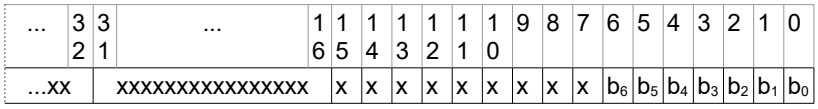
RVU.1.94	1.5 (p.8) Figure 1.1	R	All the 32-bit instructions in the base ISA have their lowest two bits (b_1b_0) set to ‘11’ and next three bits ($b_4b_3b_2$) are not equal to ‘111’.

31	...	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xxxxxxxxxxxxxxxx			x	x	x	x	x	x	x	x	x	x	x	b_4	b_3	b_2	b_1	b_0

RVU.1.95	1.5 (p.8)	H	Expanded Instruction-Length Encoding
RVU.1.96	1.5 (p.8)	I	A portion of the 32-bit instruction-encoding space has been tentatively allocated for instructions longer than 32 bits. The entirety of this space is reserved at this time, and the following proposal for encoding instructions longer than 32 bits is not considered frozen.
RVU.1.97	1.5 (p.8) Figure 1.1	T	First six bits ($b_5b_4b_3b_2b_1b_0$) of 48 bit instructions start with ‘011111’.

...	32	31	...	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
...xx		xxxxxxxxxxxxxxxx			x	x	x	x	x	x	x	x	x	x	b_5	b_4	b_3	b_2	b_1	b_0

ID	REFERENCE	TYPE	DEFINITION
RVU.1.98	1.5 (p.8) Figure 1.1	T	First seven bits ($b_6b_5b_4b_3b_2b_1b_0$) of 64 bit instructions start with '0111111'.
RVU.1.99	1.5 (p.8) Figure 1.1	T	First seven bits ($b_6b_5b_4b_3b_2b_1b_0$) of 80 to 176 bit instructions start with '1111111' and the value of the bits [14:12] ($b_{14}b_{13}b_{12}$) determine the length of the instruction: • when $b_{14}b_{13}b_{12}$ = '000' , instruction length is 80 bits • when $b_{14}b_{13}b_{12}$ = '001', instruction length is 96 bits • when $b_{14}b_{13}b_{12}$ = '010', instruction length is 112 bits • when $b_{14}b_{13}b_{12}$ = '011', instruction length is 128 bits • when $b_{14}b_{13}b_{12}$ = '100', instruction length is 144 bits • when $b_{14}b_{13}b_{12}$ = '001', instruction length is 160 bits • when $b_{14}b_{13}b_{12}$ = '011', instruction length is 176 bits
RVU.1.100	1.5 (p.8) Figure 1.1	T	Instruction encodings having first seven bits ($b_6b_5b_4b_3b_2b_1b_0$) set to '1111111' and bits [14:12] ($b_{14}b_{13}b_{12}$) set to '111' are reserved for instructions having lengths equal to or longer than 192.
RVU.1.101	1.5 (p.8, p.9)	C	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.
RVU.1.102	1.5 (p.9)	C	An implementation of the standard IMAFD ISA need only hold the most-significant 30 bits in instruction caches (a 6.25% saving). On instruction cache refills, any instructions encountered with either low bit clear should be recoded into illegal 30-bit instructions before storing in the cache to preserve illegal instruction exception behavior.
RVU.1.103	1.5 (p.9)	C	Perhaps more importantly, by condensing our base ISA into a subset of the 32-bit instruction word, we leave more space available for non-standard and custom extensions. In particular, the base RV32I ISA uses less than 1/8 of the encoding space in the 32-bit instruction word. As described in Chapter 26, an implementation that does not require support for the standard compressed instruction extension can map 3 additional non-conforming 30-bit instruction spaces into the 32-bit fixed-width format, while preserving support for standard ≥32-bit instruction-set extensions. Further, if the implementation also does not need instructions >32-bits in length, it can recover a further four major opcodes for non-conforming extensions.
RVU.1.104	1.5 (p.9)	R	(Instruction) Encodings with bits [15:0] all zeros are defined as illegal instructions.
RVU.1.105	1.5 (p.9)	R	These instructions (having first sixteen bits zero) are considered to be of minimal length: 16 bits if any 16-bit instruction-set extension is present, otherwise 32 bits.
RVU.1.106	1.5 (p.9)	R	The encoding with bits [ILEN-1:0] all ones is also illegal; this instruction is considered to be ILEN bits long.



...	32	31	...	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
...xx			xxxxxxxxxxxxxxxx		x	x	x	x	x	x	x	x	x	b_6	b_5	b_4	b_3	b_2	b_1	b_0

ID	REFERENCE	TYPE	DEFINITION
RVU.1.107	1.5 (p.9)	C	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.
RVU.1.108	1.5 (p.9)	C	Software can rely on a naturally aligned 32-bit word containing zero to act as an illegal instruction on all RISC-V implementations, to be used by software where an illegal instruction is explicitly desired. Defining a corresponding known illegal value for all ones is more difficult due to the variable-length encoding. Software cannot generally use the illegal value of ILEN bits of all 1s, as software might not know ILEN for the eventual target machine (e.g., if software is compiled into a standard binary library used by many different machines). Defining a 32-bit word of all ones as illegal was also considered, as all machines must support a 32-bit instruction size, but this requires the instruction-fetch unit on machines with ILEN>32 report an illegal instruction exception rather than access fault when such an instruction borders a protection boundary, complicating variable-instruction-length fetch and decode.
RVU.1.109	1.5 (p.9)	I	RISC-V base ISAs have either little-endian or big-endian memory systems, with the privileged architecture further defining bi-endian operation.
RVU.1.110	1.5 (p.9)	R	Instructions are stored in memory as a sequence of 16-bit little-endian parcels, regardless of memory system endianness.
RVU.1.111	1.5 (p.9)	R	Parcels forming one instruction are stored at increasing halfword addresses, with the lowest-addressed parcel holding the lowest-numbered bits in the instruction specification.
RVU.1.112	1.5 (p.9)	C	We originally chose little-endian byte ordering for the RISC-V memory system because little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). A minor point is that we have also found little-endian memory systems to be more natural for hardware designers. However, certain application areas, such as IP networking, operate on big-endian data structures, and certain legacy code bases have been built assuming big-endian processors, so we have defined big-endian and bi-endian variants of RISC-V.
RVU.1.113	1.5 (p.9, p.10)	C	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.
RVU.1.114	1.5 (p.10)	C	We further make the instruction parcels themselves little-endian to decouple the instruction encoding from the memory system endianness altogether. This design benefits both software tooling and bi-endian hardware. Otherwise, for instance, a RISC-V assembler or disassembler would always need to know the intended active endianness, despite that in bi-endian systems, the endianness mode might change dynamically during execution. In contrast, by giving instructions a fixed endianness, it is sometimes possible for carefully written software to be endianness-agnostic even in binary form, much like position-independent code.

ID	REFERENCE	TYPE	DEFINITION
RVU.1.115	1.5 (p.10)	C	The choice to have instructions be only little-endian does have consequences, however, for RISC-V software that encodes or decodes machine instructions. Big-endian JIT compilers, for example, must swap the byte order when storing to instruction memory.
RVU.1.116	1.5 (p.10)	C	Once we had decided to fix on a little-endian instruction encoding, this naturally led to placing the length-encoding bits in the LSB positions of the instruction format to avoid breaking up opcode fields.
RVU.1.117	1.6 (p.10)	H	Exceptions, Traps, and Interrupts
RVU.1.118	1.6 (p.10)	I	We use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V hart.
RVU.1.119	1.6 (p.10)	I	We use the term *interrupt* to refer to an external asynchronous event that may cause a RISC-V hart to experience an unexpected transfer of control.
RVU.1.120	1.6 (p.10)	I	We use the term *trap* to refer to the transfer of control to a trap handler caused by either an exception or an interrupt.
RVU.1.121	1.6 (p.10)	I	The instruction descriptions in following chapters describe conditions that can raise an exception during execution. The general behavior of most RISC-V EEIs is that a trap to some handler occurs when an exception is signaled on an instruction (except for floating-point exceptions, which, in the standard floating-point extensions, do not cause traps). The manner in which interrupts are generated, routed to, and enabled by a hart depends on the EEI.
RVU.1.122	1.6 (p.10)	C	Our use of “exception” and “trap” is compatible with that in the IEEE-754 floating-point standard.
RVU.1.123	1.6 (p.10)	I	How traps are handled and made visible to software running on the hart depends on the enclosing execution environment. From the perspective of software running inside an execution environment, traps encountered by a hart at runtime can have four different effects:
RVU.1.124	1.6 (p.10)	I	**Contained Trap**: The trap is visible to, and handled by, software running inside the execution environment. For example, in an EEI providing both supervisor and user mode on harts, an ECALL by a user-mode hart will generally result in a transfer of control to a supervisor-mode handler running on the same hart. Similarly, in the same environment, when a hart is interrupted, an interrupt handler will be run in supervisor mode on the hart.
RVU.1.125	1.6 (p.10)	I	**Requested Trap**: The trap is a synchronous exception that is an explicit call to the execution environment requesting an action on behalf of software inside the execution environment. An example is a system call. In this case, execution may or may not resume on the hart after the requested action is taken by the execution environment. For example, a system call could remove the hart or cause an orderly termination of the entire execution environment.
RVU.1.126	1.6 (p.11)	I	**Invisible Trap**: The trap is handled transparently by the execution environment and execution resumes normally after the trap is handled. Examples include emulating missing instructions, handling non-resident page faults in a demand-paged virtual-memory system, or handling device interrupts for a different job in a multi-programmed machine. In these cases, the software running inside the execution environment is not aware of the trap (we ignore timing effects in these definitions).

ID	REFERENCE	TYPE	DEFINITION
RVU.1.127	1.6 (p.11)	I	**Fatal Trap**: The trap represents a fatal failure and causes the execution environment to terminate execution. Examples include failing a virtual-memory page-protection check or allowing a watchdog timer to expire. Each EEI should define how execution is terminated and reported to an external environment.
RVU.1.128	1.6 (p.11) Table 1.1	I	The following table shows the characteristics of each kind of trap:

	Contained	**Requested**	**Invisible**	**Fatal**
Execution terminates?	No	No[1]	No	Yes
Software is oblivious?	No	No	Yes	Yes[2]
Handled by environment?	No	Yes	Yes	Yes

1) termination may be requested
2) imprecise fatal traps might be observable by software

RVU.1.129	1.6 (p.11)	I	The EEI defines for each trap whether it is handled precisely, though the recommendation is to maintain preciseness where possible. Contained and requested traps can be observed to be imprecise by software inside the execution environment. Invisible traps, by definition, cannot be observed to be precise or imprecise by software running inside the execution environment. Fatal traps can be observed to be imprecise by software running inside the execution environment, if known-errorful instructions do not cause immediate termination.
RVU.1.130	1.6 (p.11)	I	Because this document describes unprivileged instructions, traps are rarely mentioned. Architectural means to handle contained traps are defined in the privileged architecture manual, along with other features to support richer EEIs. Unprivileged instructions that are defined solely to cause requested traps are documented here. Invisible traps are, by their nature, out of scope for this document. Instruction encodings that are not defined here and not defined by some other means may cause a fatal trap.
RVU.1.131	1.7 (p.11)	H	UNSPECIFIED Behaviors and Values
RVU.1.132	1.7 (p.11)	I	The architecture fully describes what implementations must do and any constraints on what they may do. In cases where the architecture intentionally does not constrain implementations, the term UNSPECIFIED is explicitly used.
RVU.1.133	1.7 (p.11)	I	The term UNSPECIFIED refers to a behavior or value that is intentionally unconstrained. The definition of these behaviors or values is open to extensions, platform standards, or implementations. Extensions, platform standards, or implementation documentation may provide normative content to further constrain cases that the base architecture defines as UNSPECIFIED.
RVU.1.134	1.7 (p.12)	I	Like the base architecture, extensions should fully describe allowable behavior and values and use the term UNSPECIFIED for cases that are intentionally unconstrained. These cases may be constrained or defined by other extensions, platform standards, or implementations.

CHAPTER 2 RV32I Base Integer Instruction Set

ID	REFERENCE	TYPE	DEFINITION
RVU.2.1	2.0 (p.13)	H	RV32I Base Integer Instruction Set
RVU.2.2	2.0 (p.13) preface (p.i)	I	RV32I version is 2.1 and status is ratified.
RVU.2.3	2.0 (p.13)	C	RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might cover the ECALL/EBREAK instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE instruction as a NOP, reducing base instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).
RVU.2.4	2.0 (p.13)	C	In practice, a hardware implementation including the machine-mode privileged architecture will also require the 6 CSR instructions.
RVU.2.5	2.0 (p.13)	C	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.
RVU.2.6	2.0 (p.13)	C	Most of the commentary for RV32I also applies to the RV64I base.
RVU.2.7	2.1 (p.13)	H	Programmers' Model for Base Integer ISA
RVU.2.8	2.1 (p.13) Figure 2.1	R	For RV32I, the 32 x registers are … (There are 32 x registers)
RVU.2.9	2.1 (p.13) Figure 2.1	R	… each 32 bits wide, i.e., XLEN=32. (Register length XLEN is 32 bits)
RVU.2.10	2.1 (p.13)	I	(Registers are named as xn, where n is the address of the register, from x0 to x31)
RVU.2.11	2.1 (p.13)	R	Register x0 is hardwired with all bits equal to 0.
RVU.2.12	2.1 (p.13)	I	General purpose registers x1–x31 hold values that various instructions interpret as a collection of Boolean values, or as two's complement signed binary integers or unsigned binary integers.
RVU.2.13	2.1 (p.13)	R	There is one additional unprivileged register: the program counter pc (with a length of XLEN bits) ...
RVU.2.14	2.1 (p.13)	R	... the program counter pc holds the address of the current instruction.
RVU.2.15	2.1 (p.14)	C	There is no dedicated stack pointer or subroutine return address link register in the Base Integer ISA; the instruction encoding allows any x register to be used for these purposes
RVU.2.16	2.1 (p.14)	C	The standard software calling convention uses register x1 to hold the return address for a call
RVU.2.17	2.1 (p.14)	C	The standard software calling convention uses register x5 available as an alternate link register.
RVU.2.18	2.1 (p.14)	C	The standard calling convention uses register x2 as the stack pointer.
RVU.2.19	2.1 (p.14)	C	Hardware might choose to accelerate function calls and returns that use x1 or x5. See the descriptions of the JAL and JALR instructions.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.20	2.1 (p.14)	C	The optional compressed 16-bit instruction format is designed around the assumption that x1 is the return address register and x2 is the stack pointer. Software using other conventions will operate correctly but may have greater code size.
RVU.2.21	2.1 (p.14, p.15)	C	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.
RVU.2.22	2.1 (p.15)	C	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[§]. 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.
RVU.2.23	2.1 (p.15)	C	For resource-constrained embedded applications, we have defined the RV32E subset, which only has 16 registers (Chapter 4).
RVU.2.24	2.2 (p.15)	H	Base Instruction Formats
RVU.2.25	2.2 (p.15), Figure 2.2	R	In the base RV32I ISA, there are four core instruction formats (R/I/S/U), • R-Type: Register-Register operation format • I-Type: Register-Immediate operation format • S-Type: Store operation format • U-Type: Upper immediate operation format
RVU.2.26	2.2 (p.15)	R	All (base RV32I ISA instructions) are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory.
RVU.2.27	2.2 (p.15)	R	An *instruction-address-misaligned* exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned.
RVU.2.28	2.2 (p.15)	R	This (instruction-address-misaligned) exception is reported on the branch or jump instruction, not on the target instruction.
RVU.2.29	2.2 (p.15)	R	No instruction-address-misaligned exception is generated for a conditional branch that is not taken.
RVU.2.30	2.2 (p.15)	C	The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added (i.e., IALIGN=16)
RVU.2.31	2.2 (p.15)	C	Instruction-address-misaligned exceptions are reported on the branch or jump that would cause instruction misalignment to help debugging, and to simplify hardware design for systems with IALIGN=32, where these are the only places where misalignment can occur.

§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.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.32	2.2 (p.15)	R	The behavior upon decoding a reserved instruction is UNSPECIFIED
RVU.2.33	2.2 (p.15)	C	Some platforms may require that opcodes reserved for standard use raise an illegal-instruction exception. Other platforms may permit reserved opcode space be used for non-conforming extensions.
RVU.2.34	2.2 (p.16), Figure 2.2	R	R-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct7							rs2					rs1					funct3			rd							opcode				

RVU.2.35	2.2 (p.16), Figure 2.2	R	I-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[11:0]												rs1					funct3			rd							opcode				

RVU.2.36	2.2 (p.16), Figure 2.2	R	S-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[11:5]							rs2					rs1					funct3			imm[4:0]							opcode				

RVU.2.37	2.2 (p.16), Figure 2.2	R	U-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[31:12]																				rd							opcode				

RVU.2.38	2.2 (p.15) Figure 2.2	I	The RISC-V ISA keeps the source (*rs1* and *rs2*) (*rs1* and *rs2* fields are 5 bits address of the source registers) and destination (*rd*) registers (*rd* field is 5 bits address of the destination register) at the same position in all formats to simplify decoding.
RVU.2.39	2.2 (p.15) & 24.0 (p.129) Figure 1.1 Table 24.1	R	Opcode field encoding (opcode is the first selector for decoding the instruction that defines the category of the instruction.):

6	5	4	3	2	1	0	opcode bits
x	x	x	x	x	x	0	16 bit instruction encoding
x	x	x	x	x	0	1	16 bit instruction encoding
x	0	1	1	1	1	1	48 bit instruction encoding
0	1	1	1	1	1	1	64 bit instruction encoding
1	1	1	1	1	1	1	Greater or equal to 80 bit instruction encoding
1	x	1	1	0	1	1	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.
1	x	1	0	1	1	1	Reserved instruction
0	x	0	1	0	1	1	Custom instruction
1	1	0	1	0	1	1	Reserved instruction
others					1	1	Allocated Instruction in standard 32-bit instruction set

RVU.2.40	2.2 (p.15)	I	Except for the 5-bit immediates used in CSR instructions (Chapter 9), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.41	2.2 (p.15)	C	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).
RVU.2.42	2.2 (p.16)	C	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.
RVU.2.43	2.2 (p.16)	C	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.
RVU.2.44	2.3 (p.16)	H	Immediate Encoding Variants
RVU.2.45	2.2 (p.16) Figure 2.3	R	There are a further two variants of the instruction formats (B/J) based on the handling of immediates • B-Type: Branch operation format (Variant of S-Type format) • J-Type: Jump operation format (Variant of U-Type format)
RVU.2.46	2.3 (p.16) Figure 2.3	R	B-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	imm[10:5]							rs2				rs1						funct3		imm[4:1]		A	opcode								

A: imm[11], B: imm[12]

RVU.2.47	2.3 (p.16) Figure 2.3	R	J-Type instruction format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C	imm[10:1]										A	imm[19:12]								rd					opcode						

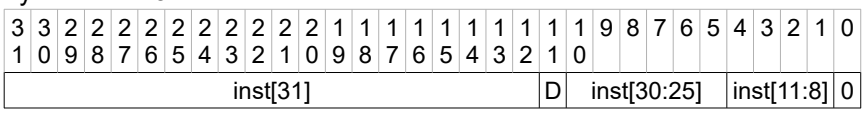
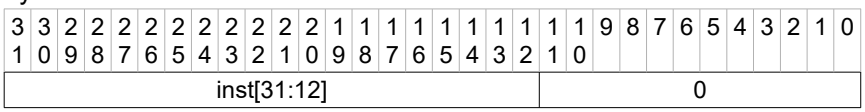
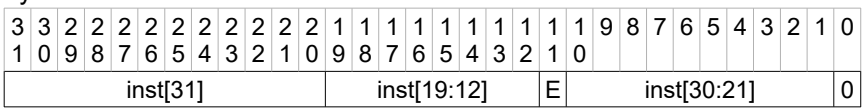
A: imm[11], C: imm[20]

RVU.2.48	2.3 (p.16)	I	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.
RVU.2.49	2.3 (p.17)	I	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.
RVU.2.50	2.3 (p.17) Figure 2.4	R	The immediate value created by I-Type instruction (I-Immediate) is a sign extended XLEN bit value having 12 bits significant value. For RV32I:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
inst[31]																				inst[31:20]											

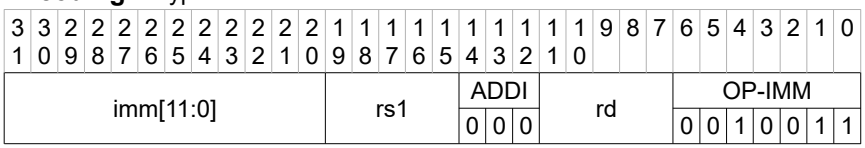
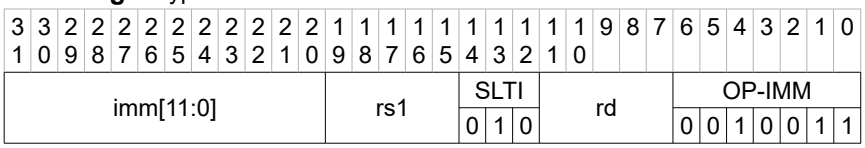
RVU.2.51	2.3 (p.17) Figure 2.4	R	The immediate value created by S-Type (S-Immediate) instruction is a sign extended XLEN bit value having 12 bits significant value. For RV32I:

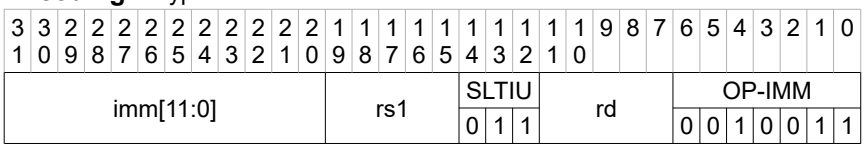
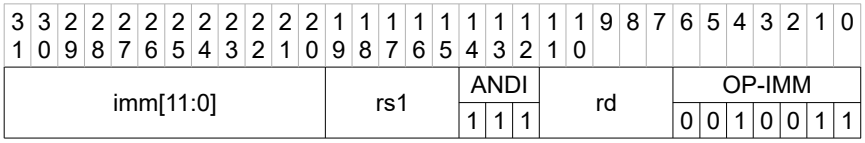
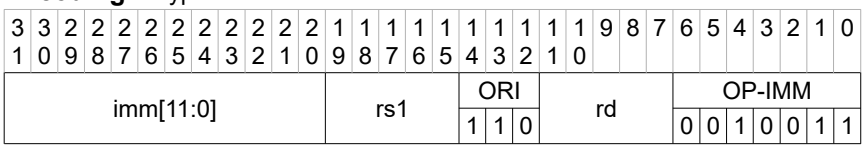
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
inst[31]																				inst[31:25]							inst[11:7]				

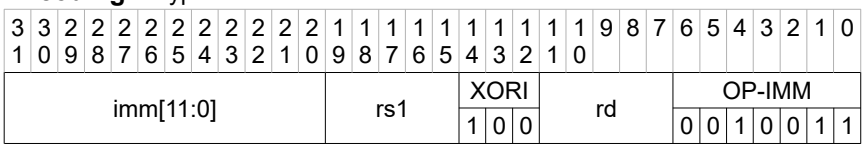
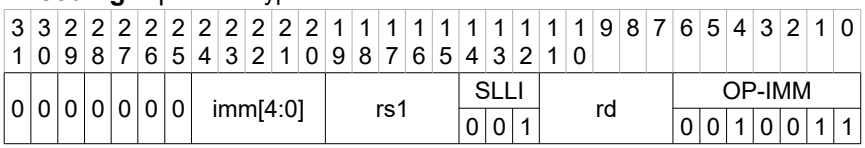
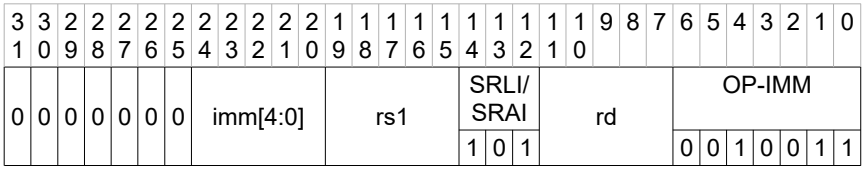
ID	REFERENCE	TYPE	DEFINITION
RVU.2.52	2.3 (p.17) Figure 2.4	R	The immediate value created by B-Type instruction (B-Immediate) is a sign extended XLEN bit value having 12 bits significant value multiplied by 2. For RV32I:  D: inst[7]
RVU.2.53	2.3 (p.17) Figure 2.4	R	The immediate value created by U-Type instruction (U-Immediate) is a sign extended XLEN bit value having 20 bits significant value multiplied by 2^{12}. For RV32I: 
RVU.2.54	2.3 (p.17) Figure 2.4	R	The immediate value created by J-Type instruction (J-Immediate) is a sign extended XLEN bit value having 20 bits significant value multiplied by 2. For RV32I:  E: inst[20]
RVU.2.55	2.3 (p.17)	C	Sign-extension is one of the most critical operations on immediates (particularly for XLEN>32), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.
RVU.2.56	2.3 (p.17)	C	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.
RVU.2.57	2.4 (p.17)	H	Integer Computational Instructions
RVU.2.58	2.4 (p.17)	I	Most integer computational instructions operate on XLEN bits of values held in the integer register file.
RVU.2.59	2.4 (p.17)	I	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.
RVU.2.60	2.4 (p.17)	R	The destination is register *rd* for both register-immediate and register-register instructions.
RVU.2.61	2.4 (p.17)	R	No integer computational instructions cause arithmetic exceptions.

ID	REFERENCE	TYPE	DEFINITION

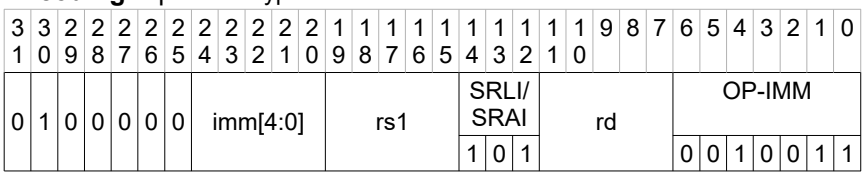
RVU.2.62	2.4 (p.17, p.18)	C	We did not include special instruction-set support for overflow checks on integer arithmetic operations in the base instruction set, as many overflow checks can be cheaply implemented using RISC-V branches. Overflow checking for unsigned addition requires only a single additional branch instruction after the addition: `add t0, t1, t2; bltu t0, t1, overflow`.
			For signed addition, if one operand's sign is known, overflow checking requires only a single branch after the addition: `addi t0, t1, +imm; blt t0, t1, overflow`. This covers the common case of addition with an immediate operand.
			For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.
			`add t0, t1, t2`
			`slti t3, t2, 0`
			`slt t4, t0, t1`
			`bne t3, t4, overflow`
			In RV64I, checks of 32-bit signed additions can be optimized further by comparing the results of ADD and ADDW on the operands.

RVU.2.63	2.4 (p.18)	H	Integer Register-Immediate Instructions
RVU.2.64	2.4 (p.18) & 24.0 (p.129, p.130)	R	ADDI Instruction Integer register-immediate signed addition operation **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1)+I-Immediate` **Explanation**: 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. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.65	2.4 (p.18)	I	`ADDI rd, rs1, 0` is used to implement the `MV rd, rs1` assembler pseudoinstruction.
RVU.2.66	2.4 (p.18) & 24.0 (p.129, p.130)	R	SLTI Instruction Set less than immediate operation with signed comparison **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=1 if x(rs1)<I-Immediate; x(rd)=0 otherwise` **Explanation**: 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*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.67	2.4 (p.18) & 24.0 (p.129, p.130)	R	SLTIU Instruction Set less than immediate operation with unsigned comparison **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=1 if x(rs1)<I-Immediate; x(rd)=0 otherwise` **Explanation**: SLTIU (set less than immediate unsigned) places the value 1 in register *rd* if register *rs1* which is treated as an unsigned integer is less than the immediate which is sign extended first and then treated as an unsigned integer, else 0 is written to *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.68	2.4 (p.18)	I	Note, `SLTIU rd, rs1, 1` sets *rd* to 1 if *rs1* equals zero, otherwise sets *rd* to 0 (assembler pseudoinstruction `SEQZ rd, rs`).
RVU.2.69	2.4 (p.18) & 24.0 (p.129, p.130)	R	ANDI Instruction Integer register-immediate bit wise and operation **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) and I-Immediate` **Explanation**: ANDI is logical operation that perform bitwise AND on register *rs1* and the sign-extended immediate and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.70	2.4 (p.18) & 24.0 (p.129, p.130)	R	ORI Instruction Integer register-immediate bit wise or operation **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) or I-Immediate` **Explanation**: ORI is logical operation that perform bitwise OR on register *rs1* and the sign-extended immediate and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

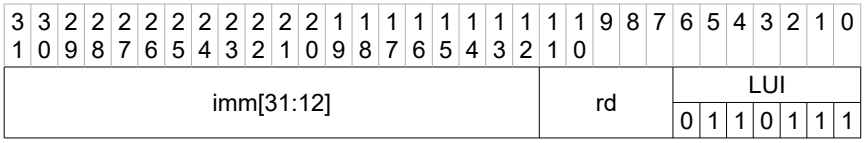
ID	REFERENCE	TYPE	DEFINITION
RVU.2.71	2.4 (p.18) & 24.0 (p.129, p.130)	R	XORI Instruction Integer register-immediate bit wise xor operation **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) xor I-Immediate` **Explanation**: XORI is logical operation that perform bitwise XOR on register *rs1* and the sign-extended immediate and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.72	2.4 (p.18)	I	Note, `XORI rd, rs1, -1` performs a bitwise logical inversion of register *rs1* (assembler pseudoinstruction `NOT rd, rs`).
RVU.2.73	2.4 (p.18) & 24.0 (p.129, p.130)	R	SLLI Instruction Logical left shift register by immediate operation **Encoding**: Special I-Type  **Valid Base**: RV32 **Task**: `x(rd)=x(rs1)<<I-Immediate[4:0]` **Explanation**: SLLI is a logical left shift operation (zeros are shifted into the lower bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31). Higher order bits of I-Immediate (I-Immediate[11:5]) are expected to be zero. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.74	2.4 (p.18, p.19) & 24.0 (p.129, p.130)	R	SRLI Instruction Logical right shift register by immediate operation **Encoding**: Special I-Type  **Valid Base**: RV32 **Task**: `x(rd)=x(rs1)>>I-Immediate[4:0]` **Explanation**: SRLI is a logical right shift operation (zeros are shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31). The value of I-Immediate[10] bit determines if this is a logical or an arithmetic shift. For SRLI, I-Immediate[10]=0. Other higher order bits of I-Immediate are expected to be zero. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.75	2.4 (p.18, p.19) & 24.0 (p.129, p.130)	R	SRAI Instruction Arithmetic right shift register by immediate operation **Encoding**: Special I-Type



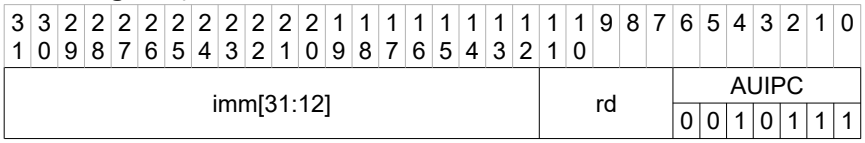
Valid Base: RV32
Task: `x(rd)=x(rs1)>>>I-Immediate[4:0]`
Explanation: SRAI is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31).
The value of I-Immediate[10] bit determines if this is a logical or arithmetic shift. For SRAI, I-Immediate[10]=1.
Other higher order bits of I-Immediate are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.2.76	2.4 (p.19) & 24.0 (p.129, p.130)	R	LUI Instruction Load upper immediate operation **Encoding**: U-Type

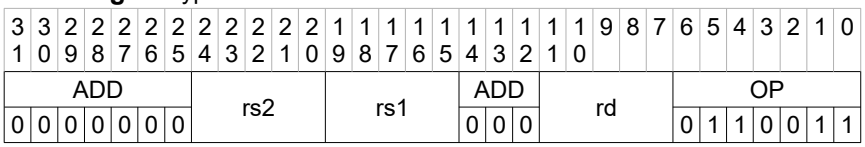
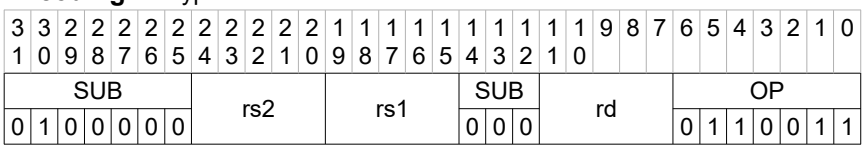


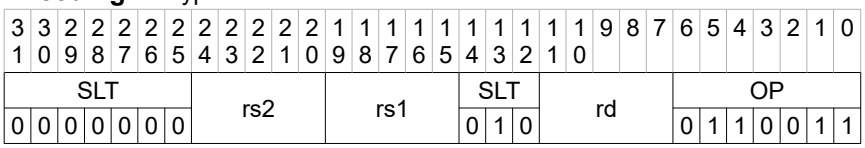
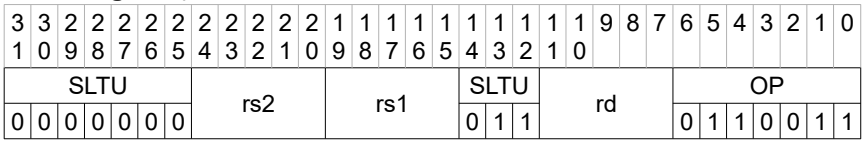
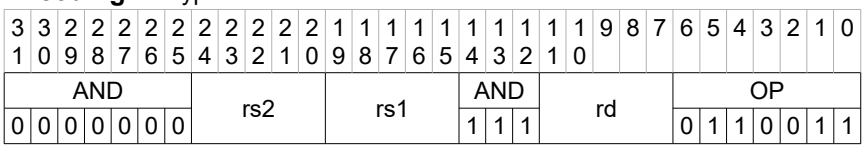
Valid Base: RV32
Task: `x(rd)=U-Immediate[31:0]`
Explanation: LUI places the U-Immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

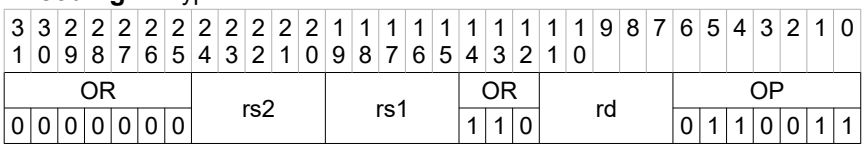
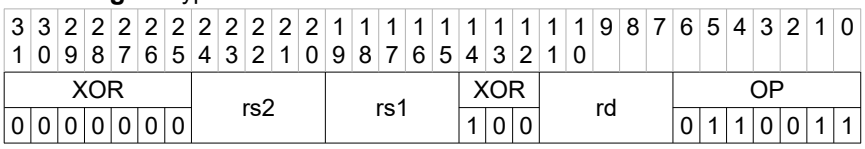
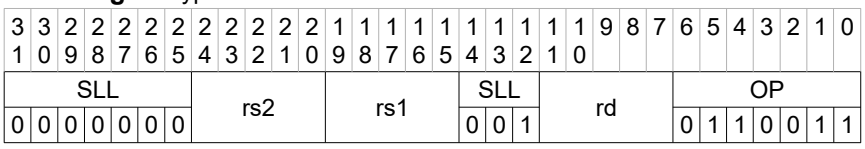
RVU.2.77	2.4 (p.19) & 24.0 (p.129, p.130)	R	AUIPC Instruction Add upper immediate to PC operation **Encoding**: U-Type

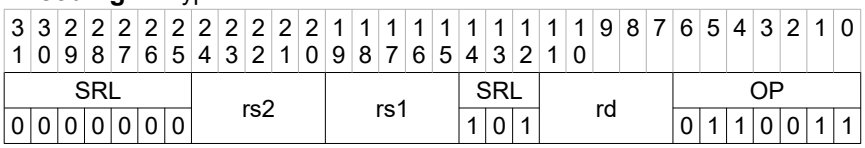
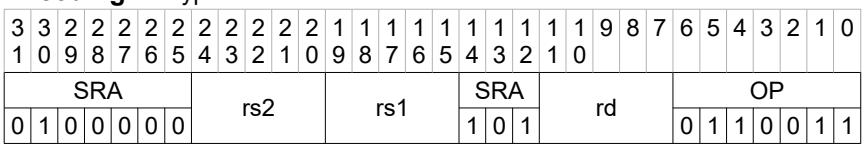


Valid Base: RV32
Task: `x(rd)=PC+U-Immediate[31:0]`
Explanation: AUIPC forms a 32-bit offset from the 20-bit U-Immediate, filling in the lowest 12 bits with zeros, adds this offset to the address of the AUIPC instruction (determined from PC), then places the result in register *rd*.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

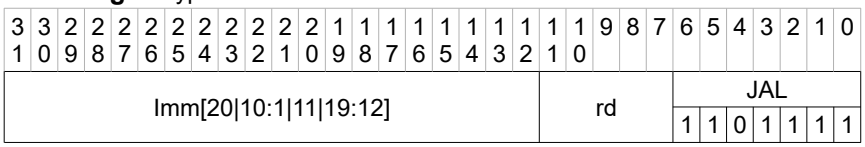
ID	REFERENCE	TYPE	DEFINITION
RVU.2.78	2.4 (p.19)	C	The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative 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.
RVU.2.79	2.4 (p.19)	C	The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the local PC (of the instruction following the JAL), it might cause pipeline breaks in simpler microarchitectures or pollute BTB structures in more complex microarchitectures.
RVU.2.80	2.4 (p.19)	H	Integer Register-Register Operations
RVU.2.81	2.4 (p.19)	I	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.
RVU.2.82	2.4 (p.19) & 24.0 (p.129, p.130)	R	ADD Instruction Integer register-register signed addition operation **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1)+x(rs2)` **Explanation**: ADD performs the addition of *rs1* and *rs2* and writes the result into register *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.83	2.4 (p.19) & 24.0 (p.129, p.130)	R	SUB Instruction Integer register-register signed subtraction operation **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1)-x(rs2)` **Explanation**: SUB performs the subtraction of *rs2* from *rs1* and writes the result into register *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.84	2.4 (p.19) & 24.0 (p.129, p.130)	R	SLT Instruction Set less than register signed comparison **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=1 if x(rs1)<x(rs2); x(rd)=0 otherwise` **Explanation**: SLT operation places the value 1 in register *rd* if register *rs1* is less than the register *rs2* when both are treated as signed numbers, else 0 is written to *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.85	2.4 (p.19) & 24.0 (p.129, p.130)	R	SLTU Instruction Set less than register unsigned comparison **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=1 if x(rs1)<x(rs2); x(rd)=0 otherwise` **Explanation**: SLTU operation places the value 1 in register *rd* if register *rs1* is less than the register *rs2* when both are treated as unsigned numbers, else 0 is written to *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.86	2.4 (p.20)	I	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`).
RVU.2.87	2.4 (p.20) & 24.0 (p.129, p.130)	R	AND Instruction Integer register-register bit wise and operation **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) and x(rs2)` **Explanation**: AND is logical operation that performs bitwise AND on register *rs1* and register *rs2* and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.88	2.4 (p.20) & 24.0 (p.129, p.130)	R	OR Instruction Integer register-register bit wise and operation **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) or x(rs2)` **Explanation**: OR is logical operation that performs bitwise OR on register *rs1* and register *rs2* and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.89	2.4 (p.20) & 24.0 (p.129, p.130)	R	XOR Instruction Integer register-register bit wise xor operation **Encoding**: R-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=x(rs1) xor x(rs2)` **Explanation**: XOR is logical operation that performs bitwise XOR on register *rs1* and register *rs2* and place the result in *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.90	2.4 (p.20) & 24.0 (p.129, p.130)	R	SLL Instruction Logical left shift register by register operation **Encoding**: R-Type  **Valid Base**: RV32 **Task**: `x(rd)=x(rs1)<<x(rs2)[4:0]` **Explanation**: SLL is a logical left shift operation (zeros are shifted into the lower bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31). **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

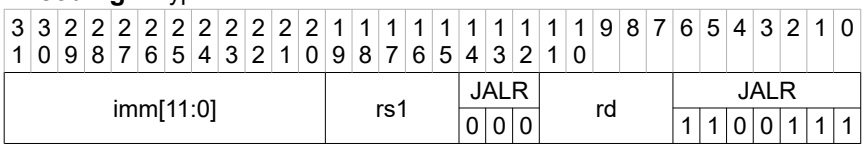
ID	REFERENCE	TYPE	DEFINITION
RVU.2.91	2.4 (p.20) & 24.0 (p.129, p.130)	R	SRL Instruction Logical right shift register by register operation **Encoding**: R-Type  **Valid Base**: RV32 **Task**: `x(rd)=x(rs1)>>x(rs2)[4:0]` **Explanation**: SRL is a logical right shift operation (zeros are shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31). **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.92	2.4 (p.20) & 24.0 (p.129, p.130)	R	SRA Instruction Arithmetic right shift register by register operation **Encoding**: R-Type  **Valid Base**: RV32 **Task**: `x(rd)=x(rs1)>>>x(rs2)[4:0]` **Explanation**: SRA is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31). **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.2.93	2.4 (p.20)	H	NOP Instruction
RVU.2.94	2.4 (p.20)	R	The NOP instruction does not change any architecturally visible state, except for advancing the pc and incrementing any applicable performance counters.
RVU.2.95	2.4 (p.20)	R	NOP is encoded as `ADDI x0, x0, 0`.
RVU.2.96	2.4 (p.20)	C	NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as for more readable disassembly output. The other NOP encodings are made available for HINT instructions (Section 2.9).
RVU.2.97	2.4 (p.20)	C	ADDI was chosen for the NOP encoding as this is most likely to take fewest resources to execute across a range of systems (if not optimized away in decode). In particular, the instruction only reads one register. Also, an ADDI functional unit is more likely to be available in a superscalar design as adds are the most common operation. In particular, address-generation functional units can execute ADDI using the same hardware needed for base+offset address calculations, while register-register ADD or logical/shift operations require additional hardware.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.98	2.5 (p.20)	H	Control Transfer Instructions
RVU.2.99	2.5 (p.20)	I	RV32I provides two types of control transfer instructions: unconditional jumps and conditional branches.
RVU.2.100	2.5 (p.20)	R	Control transfer instructions (JAL, JALR, BEQ, BNE, BLT, BLTU, BGE, BGEU) in RV32I do not have architecturally visible delay slots
RVU.2.101	2.5 (p.20)	H	Unconditional Jumps
RVU.2.102	2.5 (p.20) & 24.0 (p.129, p.130)	R	JAL Instruction Unconditional jump and link instruction **Encoding**: J-Type



Valid Base: RV32, RV64, RV128
Task: `x(rd)=PC+4; PC=PC+J-Immediate`
Explanation: The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. Jumps can therefore target a ±1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register *rd*.
Special Case: If *rd* is zero (addressing x0), return address is not saved but jump taken
Exception: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.

RVU.2.103	2.5 (p.20)	I	The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.
RVU.2.104	2.5 (p.20)	C	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.
RVU.2.105	2.5 (p.21)	I	Plain unconditional jumps (assembler pseudoinstruction `J`) are encoded as a `JAL` with `rd=x0`.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.106	2.5 (p.21) & 24.0 (p.129, p.130)	R	JALR Instruction Indirect jump and link instruction **Encoding**: I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd)=PC+4; PC=(x(rs1)+I-Immediate) and FFFFFFFE` **Explanation**: 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*. **Special Case**: If rd is zero (addressing x0), return address is not saved but jump taken. **Exception**: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary.
RVU.2.107	2.5 (p.21, p.23)	C	Instruction-address-misaligned exceptions are not possible on machines that support extensions with 16-bit aligned instructions, such as the compressed instruction-set extension, C.
RVU.2.108	2.5 (p.21)	C	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.
RVU.2.109	2.5 (p.21)	C	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. Alternatively, an ABI could dedicate a general-purpose register to point to a library elsewhere in the address space.
RVU.2.110	2.5 (p.21)	I	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.

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.2.111 2.5 (p.21, p.22) Table 2.1 R 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 below

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

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.

RVU.2.112 2.5 (p.22) C 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)`

RVU.2.113 2.5 (p.22) H Conditional Branches

RVU.2.114 2.5 (p.22) I All branch instructions use the B-type instruction format.

RVU.2.115 2.5 (p.22) I The 12-bit B-immediate encodes signed offsets in multiples of 2 bytes.

RVU.2.116 2.5 (p.22) I The offset is sign-extended and added to the address of the branch instruction to give the target address. The conditional branch range is ±4 KiB.

RVU.2.117 2.5 (p.22, p.23) & 24.0 (p.129, p.130) R BEQ Instruction
Branch if equal operation
Encoding: B-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[12|10:5]							rs2					rs1					BEQ			imm[4:1|11]				BRANCH							
																	0	0	0						1	1	0	0	0	1	1

Valid Base: RV32, RV64, RV128
Task: `PC=(PC+B-Immediate) if x(rs1)=x(rs2); PC=PC+4 otherwise`
Explanation: BEQ takes the branch if registers *rs1* and *rs2* are equal. The branch address is calculated by adding B-Immediate to the address of the branch instruction.
Special Case: none
Exception: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.118	2.5 (p.22, p.23) & 24.0 (p.129, p.130)	R	BNE Instruction Branch if not equal instruction **Encoding**: B-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[12|10:5]							rs2					rs1				BNE			imm[4:1|11]			BRANCH									
																0	0	1				1	1	0	0	0	1	1			

Valid Base: RV32, RV64, RV128

Task:
```
PC=(PC+B-Immediate) if x(rs1)!=x(rs2);
PC=PC+4 otherwise
```

Explanation: BNE takes the branch if registers *rs1* and *rs2* are not equal. The branch address is calculated by adding B-Immediate to the address of the branch instruction.

Special Case: none

Exception: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.119	2.5 (p.22, p.23) & 24.0 (p.129, p.130)	R	BLT Instruction Branch if less than signed operation **Encoding**: B-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[12|10:5]							rs2					rs1				BLT			imm[4:1|11]			BRANCH									
																1	0	0				1	1	0	0	0	1	1			

Valid Base: RV32, RV64, RV128

Task:
```
PC=(PC+B-Immediate) if x(rs1)<x(rs2);
PC=PC+4 otherwise
```

Explanation: BLT takes the branch if register *rs1* is less than *rs2* when both registers are treated as signed integer values. The branch address is calculated by adding B-Immediate to the address of the branch instruction.

Special Case: none

Exception: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.120	2.5 (p.22, p.23) & 24.0 (p.129, p.130)	R	BLTU Instruction Branch if less than unsigned operation **Encoding**: B-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[12|10:5]							rs2					rs1				BLTU			imm[4:1|11]			BRANCH									
																1	1	0				1	1	0	0	0	1	1			

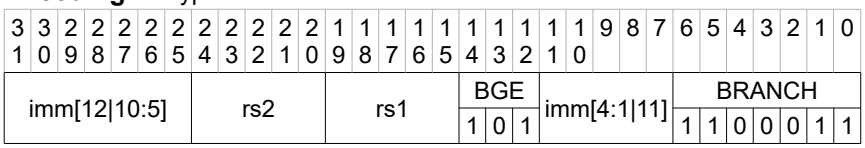
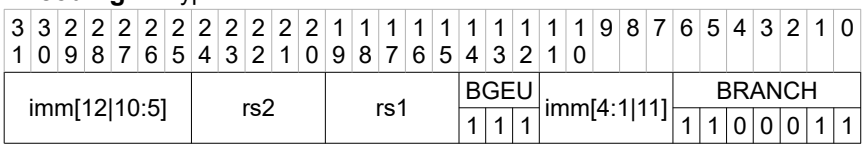
Valid Base: RV32, RV64, RV128

Task:
```
PC=(PC+B-Immediate) if x(rs1)<x(rs2);
PC=PC+4 otherwise
```

Explanation: BLT takes the branch if register *rs1* is less than *rs2* when both registers are treated as unsigned integer values. The branch address is calculated by adding B-Immediate to the address of the branch instruction.

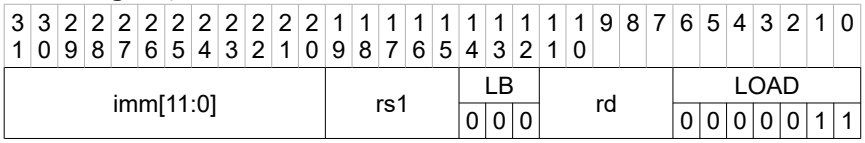
Special Case: none

Exception: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.121	2.5 (p.22) & 24.0 (p.129, p.130)	R	BGE Instruction Branch if greater than or equal to signed operation **Encoding**: B-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `PC=(PC+B-Immediate) if x(rs1)≥x(rs2);` `PC=PC+4 otherwise` **Explanation**: BGE takes the branch if register *rs1* is greater than or equal to *rs2* when both registers are treated as signed integer values. The branch address is calculated by adding B-Immediate to the address of the branch instruction. **Special Case**: none **Exception**: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.
RVU.2.122	2.5 (p.22) & 24.0 (p.129, p.130)	R	BGEU Instruction Branch if greater than or equal to unsigned operation **Encoding**: B-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `PC=(PC+B-Immediate) if x(rs1)≥x(rs2);` `PC=PC+4 otherwise` **Explanation**: BGEU takes the branch if register *rs1* is greater than or equal to *rs2* when both registers are treated as unsigned integer values. The branch address is calculated by adding B-Immediate to the address of the branch instruction. **Special Case**: none **Exception**: It will generate an instruction-address-misaligned exception if the target address is not aligned to a four-byte boundary and the branch condition evaluates to true.
RVU.2.123	2.5 (p.22)	I	Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.
RVU.2.124	2.5 (p.22)	C	Signed array bounds may be checked with a single BLTU instruction, since any negative index will compare greater than any nonnegative bound.
RVU.2.125	2.5 (p.22)	O	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.
RVU.2.126	2.5 (p.22)	O	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.
RVU.2.127	2.5 (p.22)	O	Dynamic predictors should quickly learn any predictable branch behavior.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.128	2.5 (p.22)	O	Unlike some other architectures, the RISC-V jump (JAL with *rd*=x0) instruction should always be used for unconditional branches instead of a conditional branch instruction with an always true condition. RISC-V jumps are also PC-relative and support a much wider offset range than branches, and will not pollute conditional-branch prediction tables.
RVU.2.129	2.5 (p.23)	C	The conditional branches were designed to include arithmetic comparison operations between two registers (as also done in PA-RISC, Xtensa, and MIPS R6), rather than use condition codes (x86, ARM, SPARC, PowerPC), or to only compare one register against zero (Alpha, MIPS), or two registers only for equality (MIPS). This design was motivated by the observation that a combined compare-and-branch instruction fits into a regular pipeline, avoids additional condition code state or use of a temporary register, and reduces static code size and dynamic instruction fetch traffic. Another point is that comparisons against zero require non-trivial circuit delay (especially after the move to static logic in advanced processes) and so are almost as expensive as arithmetic magnitude compares. Another advantage of a fused compare-and-branch instruction is that branches are observed earlier in the front-end instruction stream, and so can be predicted earlier. There is perhaps an advantage to a design with condition codes in the case where multiple branches can be taken based on the same condition codes, but we believe this case to be relatively rare.
RVU.2.130	2.5 (p.23)	C	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.
RVU.2.131	2.5 (p.23)	C	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.
RVU.2.132	2.5 (p.23)	C	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 and have been implemented in commercial processors. 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.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.133	2.6 (p.24)	H	Load and Store Instructions
RVU.2.134	2.6 (p.24)	R	RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.
RVU.2.135	2.6 (p.24)	R	RV32I provides a 32-bit address space that is byte-addressed
RVU.2.136	2.6 (p.24)	I	The EEI will define what portions of the address space are legal to access with which instructions (e.g., some addresses might be read only, or support word access only).
RVU.2.137	2.6 (p.24)	R	Loads with a destination of x0 must still raise any exceptions and cause any other side effects even though the load value is discarded.
RVU.2.138	2.6 (p.24)	R	The EEI will define whether the memory system is little-endian or big-endian. In RISC-V, endianness is byte-address invariant.
RVU.2.139	2.6 (p.24)	C	In a system for which endianness is byte-address invariant, the following property holds: if a byte is stored to memory at some address in some endianness, then a byte-sized load from that address in any endianness returns the stored value. In a little-endian configuration, multibyte stores write the least-significant register byte at the lowest memory byte address, followed by the other register bytes in ascending order of their significance. Loads similarly transfer the contents of the lesser memory byte addresses to the less-significant register bytes. In a big-endian configuration, multibyte stores write the most-significant register byte at the lowest memory byte address, followed by the other register bytes in descending order of their significance. Loads similarly transfer the contents of the greater memory byte addresses to the less-significant register bytes.
RVU.2.140	2.6 (p.24) & 24.0 (p.129, p.130)	R	LB Instruction Load signed byte from memory to register operation **Encoding**: I-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[11:0]												rs1					LB			rd					LOAD							
																	0	0	0			0	0	0	0	0	0	1	1		

Valid Base: RV32

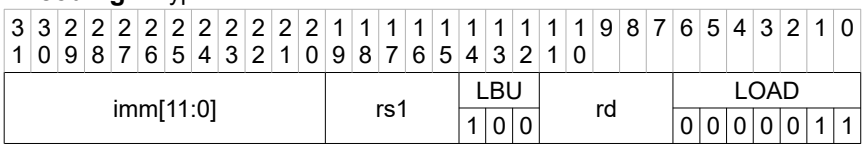
Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr)
x(rd)[31:8]=mem_val[7]; x(rd)[7:0]=mem_val
```

Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one byte (8-bit) value from memory at the effective address, then sign-extends to 32-bits and stores it into *rd*.

Special Case: If *rd* is zero (addressing x0), mem_val is not saved but read is done.

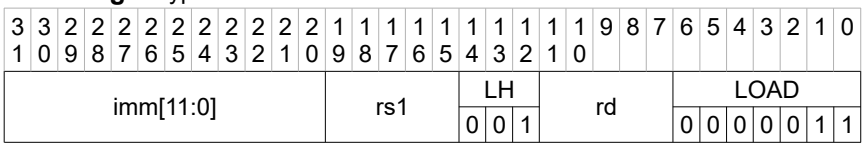
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.141	2.6 (p.24) & 24.0 (p.129, p.130)	R	LBU Instruction Load unsigned byte from memory to register operation **Encoding**: I-Type



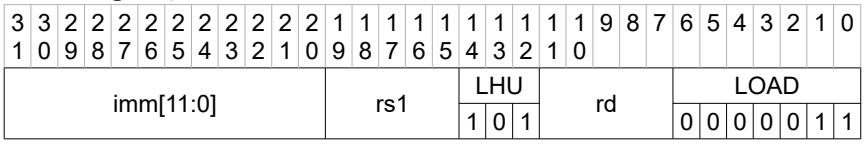
Valid Base: RV32
Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr)
x(rd)[31:8]=0; x(rd)[7:0]=mem_val
```
Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one byte (8-bit) value from memory at the effective address, then zero-extends to 32-bits and stores it into *rd*.
Special Case: If *rd* is zero (addressing x0), mem_val is not saved but read is done.
Exception: none

RVU.2.142	2.6 (p.24) & 24.0 (p.129, p.130)	R	LH Instruction Load signed 2 bytes from memory to register operation **Encoding**: I-Type



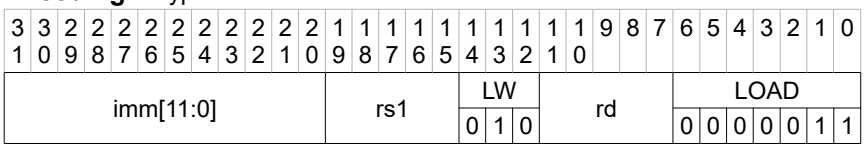
Valid Base: RV32
Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr:mem_addr+1)
x(rd)[31:16]=mem_val[15]; x(rd)[15:0]=mem_val
```
Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 2 bytes (16-bit) value from memory at the effective address, then sign-extends to 32-bits and stores it into *rd*.
Special Case: If *rd* is zero (addressing x0), mem_val is not saved but read is done.
Exception: none

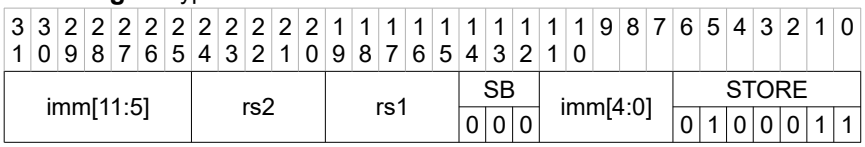
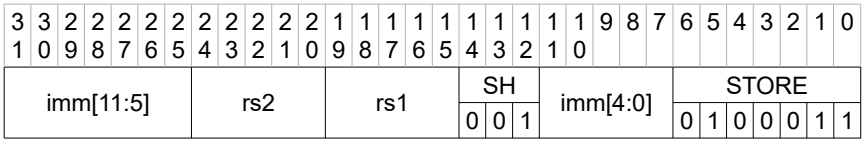
RVU.2.143	2.6 (p.24) & 24.0 (p.129, p.130)	R	LHU Instruction Load unsigned 2 bytes from memory to register operation **Encoding**: I-Type

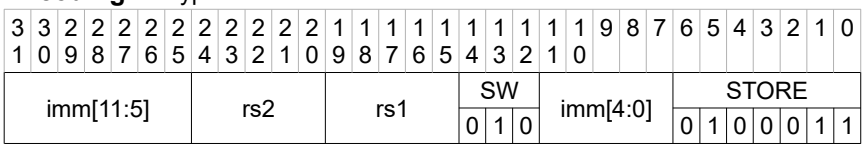


Valid Base: RV32
Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr:mem_addr+1)
x(rd)[31:16]=0; x(rd)[15:0]=mem_val
```
Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 2 bytes (16-bit) value from memory at the effective address, then zero-extends to 32-bits and stores it into *rd*.
Special Case: If *rd* is zero (addressing x0), mem_val is not saved but read is done.
Exception: none

ID	REFERENCE	TYPE	DEFINITION

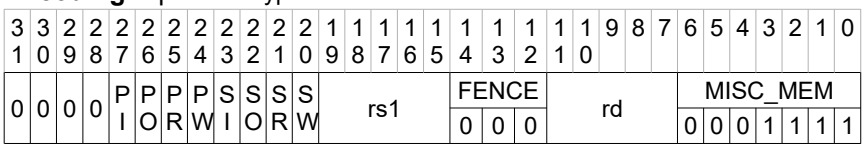
RVU.2.144	2.6 (p.24) & 24.0 (p.129, p.130)	R	LW Instruction
			Load 4 bytes from memory to register operation
			Encoding: I-Type
			 
			Valid Base: RV32
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+3)`
			`x(rd)=mem_val`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 4 bytes (32-bit) value from memory at the effective address and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0), mem_val is not saved but read is done.
			Exception: none

RVU.2.145	2.6 (p.24) & 24.0 (p.129, p.130)	R	SB Instruction Store a byte from register to memory operation **Encoding**: S-Type  **Valid Base**: RV32 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr)=x(rs2)[7:0]` **Explanation**: The memory address is obtained by adding register *rs1* to the S-Immediate value. It stores the low order byte of *rs2* in to the memory at the effective address. **Special Case**: none **Exception**: none
RVU.2.146	2.6 (p.24) & 24.0 (p.129, p.130)	R	SH Instruction Store 2 bytes from register to memory operation **Encoding**: S-Type  **Valid Base**: RV32 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr:mem_addr+1)=x(rs2)[15:0]` **Explanation**: The memory address is obtained by adding register *rs1* to the S-Immediate value. It stores the low order 2 bytes of *rs2* in to the memory at the effective address.. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.2.147	2.6 (p.24) & 24.0 (p.129, p.130)	R	SW Instruction Store 4 bytes from register to memory operation **Encoding**: S-Type  **Valid Base**: RV32 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr:mem_addr+3)=x(rs2)[31:0]` **Explanation**: The memory address is obtained by adding register *rs1* to the S-Immediate value. It stores the contents of *rs2* (32 bits) in to the memory at the effective address. **Special Case**: none **Exception**: none
RVU.2.148	2.6 (p.24)	R	Regardless of EEI, loads and stores whose effective addresses are naturally aligned shall not raise an address-misaligned exception.
RVU.2.149	2.6 (p.24, p.25)	R	Loads and stores where the effective address is not naturally aligned to the referenced datatype (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses) have behavior dependent on the EEI.
RVU.2.150	2.6 (p.25)	O	An EEI may guarantee that misaligned loads and stores are fully supported, and so the software running inside the execution environment will never experience a contained or fatal address-misaligned trap. In this case, the misaligned loads and stores can be handled in hardware, or via an invisible trap into the execution environment implementation, or possibly a combination of hardware and invisible trap depending on address.
RVU.2.151	2.6 (p.25)	O	An EEI may not guarantee misaligned loads and stores are handled invisibly. In this case, loads and stores that are not naturally aligned may either complete execution successfully or raise an exception. The exception raised can be either an address-misaligned exception or an access-fault exception. For a memory access that would otherwise be able to complete except for the misalignment, an access exception can be raised instead of an address-misaligned exception if the misaligned access should not be emulated, e.g., if accesses to the memory region have side effects. When an EEI does not guarantee misaligned loads and stores are handled invisibly, the EEI must define if exceptions caused by address misalignment result in a contained trap (allowing software running inside the execution environment to handle the trap) or a fatal trap (terminating execution).

ID	REFERENCE	TYPE	DEFINITION
RVU.2.152	2.6 (p.25)	C	Misaligned accesses are occasionally required when porting legacy code, and help performance on applications when using any form of packed-SIMD extension or handling externally packed data structures. Our rationale for allowing EEIs to choose to support misaligned accesses via the regular load and store instructions is to simplify the addition of misaligned hardware support. One option would have been to disallow misaligned accesses in the base ISA and then provide some separate ISA support for misaligned accesses, either special instructions to help software handle misaligned accesses or a new hardware addressing mode for misaligned accesses. Special instructions are difficult to use, complicate the ISA, and often add new processor state (e.g., SPARC VIS align address offset register) or complicate access to existing processor state (e.g., MIPS LWL/LWR partial register writes). In addition, for loop-oriented packed-SIMD code, the extra overhead when operands are misaligned motivates software to provide multiple forms of loop depending on operand alignment, which complicates code generation and adds to loop startup overhead. New misaligned hardware addressing modes take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).
RVU.2.153	2.6 (p.25)	R	Even when misaligned loads and stores complete successfully, these accesses might run extremely slowly depending on the implementation (e.g., when implemented via an invisible trap). Furthermore, whereas naturally aligned loads and stores are guaranteed to execute atomically, misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.
RVU.2.154	2.6 (p.25)	C	We do not mandate atomicity for misaligned accesses so execution environment implementations can use an invisible machine trap and a software handler to handle some or all misaligned accesses. If hardware misaligned support is provided, software can exploit this by simply using regular load and store instructions. Hardware can then automatically optimize accesses depending on whether runtime addresses are aligned.
RVU.2.155	2.7 (p.26)	H	Memory Ordering Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.2.156	2.7 (p.26) & 24.0 (p.129, p.130)	R	FENCE Instruction Order device I/O & Memory Access **Encoding**: Special I-Type

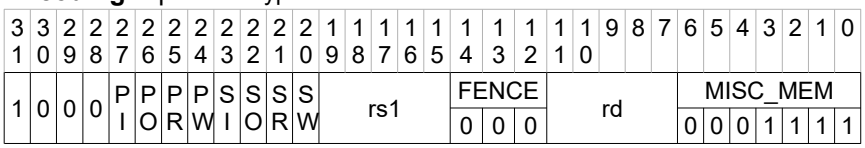


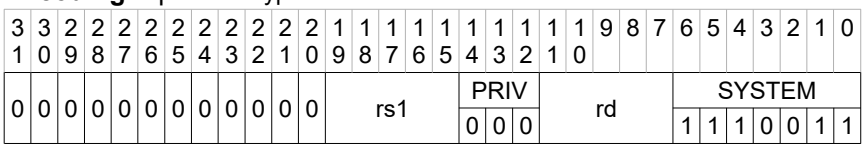
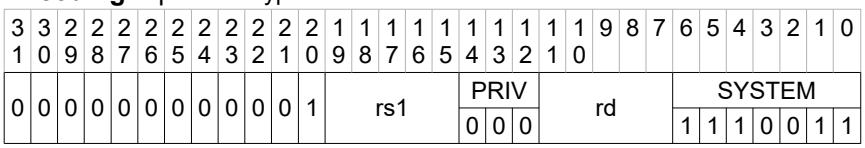
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	PI	PO	PR	PW	SI	SO	SR	SW	rs1					FENCE			rd							MISC_MEM					
																	0	0	0								0	0	0	1	1	1	1

Valid Base: RV32, RV64, RV128
Task: –
Explanation:
PI: Predecessor Input operation for I/O memory space
PO: Predecessor Output operation for I/O memory space
PR: Predecessor Read operation for regular memory space
PW: Predecessor Write operation for regular memory space
SI: Successor Input operation for I/O memory space
SO: Successor Output operation for I/O memory space
SR: Successor Read operation for regular memory space
SW: Successor Write operation for regular memory space

The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISCV harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same.
FENCE orders all memory operations in its *predecessor* set before all memory operations in its *successor* set.
Special Case: none
Exception: none

RVU.2.157	2.7 (p.26)	R	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.
RVU.2.158	2.7 (p.26)	I	Chapter 14 provides a precise description of the RISC-V memory consistency model.
RVU.2.159	2.7 (p.26)	I	The EEI will define what I/O operations are possible, and in particular, which memory addresses when accessed by load and store instructions will be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new I/O instructions that will also be ordered using the I and O bits in a FENCE.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.160	2.7 (p.26) & 24.0 (p.129, p.130)	R	FENCE.TSO Instruction Order device I/O & Memory Access **Encoding**: Special I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: – **Explanation**: The optional FENCE.TSO instruction is encoded as a FENCE instruction with *predecessor*=RW, and *successor*=RW. FENCE.TSO orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the FENCE.TSO's predecessor set unordered with non-AMO loads in its successor set. **Special Case**: none **Exception**: none
RVU.2.161	2.7 (p.26)	C	The FENCE.TSO encoding was added as an optional extension to the original base FENCE instruction encoding. The base definition requires that implementations ignore any set bits and treat the FENCE as global, and so this is a backwards-compatible extension.
RVU.2.162	2.7 (p.26)	R	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.
RVU.2.163	2.7 (p.27)	C	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.
RVU.2.164	2.8 (p.27)	H	Environment Call and Breakpoints
RVU.2.165	2.8 (p.27)	I	SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in Chapter 9, and the base unprivileged instructions are described in the following section.
RVU.2.166	2.8 (p.27)	C	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.167	2.8 (p.27) & 24.0 (p.129, p.130)	R	ECALL Instruction Environment service call **Encoding**: Special I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: – **Explanation**: It causes a precise requested trap to the supporting execution environment. The ECALL instruction is used to make a service request to the execution environment. The EEI will define how parameters for the service request are passed, but usually these will be in defined locations in the integer register file. **Special Case**: none **Exception**: none
RVU.2.168	2.8 (p.27) & 24.0 (p.129, p.130)	R	EBREAK Instruction Environment break **Encoding**: Special I-Type  **Valid Base**: RV32, RV64, RV128 **Task**: – **Explanation**: It causes a precise requested trap to the supporting execution environment. The EBREAK instruction is used to return control to a debugging environment. **Special Case**: none **Exception**: none
RVU.2.169	2.8 (p.27)	C	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.
RVU.2.170	2.8 (p.27)	C	EBREAK was primarily designed to be used by a debugger to cause execution to stop and fall back into the debugger. EBREAK is also used by the standard gcc compiler to mark code paths that should not be executed.

RVU.2.171	2.8 (p.27, p.28)	C	Another use of EBREAK is to support “semihosting”, where the execution environment includes a debugger that can provide services over an alternate system call interface built around the EBREAK instruction. Because the RISC-V base ISA does not provide more than one EBREAK instruction, RISC-V semihosting uses a special sequence of instructions to distinguish a semihosting EBREAK from a debugger inserted EBREAK.
			`slli x0, x0, 0x1f # Entry NOP`
			`ebreak # Break to debugger`
			`srai x0, x0, 7 # NOP encoding the semihosting call number 7`
			Note that these three instructions must be 32-bit-wide instructions, i.e., they mustn’t be among the compressed 16-bit instructions described in Chapter 16.

ID	REFERENCE	TYPE	DEFINITION
RVU.2.172	2.8 (p.28)	C	The shift NOP instructions are still considered available for use as HINTS. Semihosting is a form of service call and would be more naturally encoded as an ECALL using an existing ABI, but this would require the debugger to be able to intercept ECALLs, which is a newer addition to the debug standard. We intend to move over to using ECALLs with a standard ABI, in which case, semihosting can share a service ABI with an existing standard. We note that ARM processors have also moved to using SVC instead of BKPT for semihosting calls in newer designs.
RVU.2.173	2.9 (p.28)	H	HINT Instructions
RVU.2.174	2.9 (p.28)	I	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.
RVU.2.175	2.9 (p.28)	C	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.
RVU.2.176	2.9 (p.28) Table 2.3	R	Following HINT instructions are reserved for standard HINTs. but none are presently defined.

Instruction	Constraints	Code Points
LUI	rd=x0	2^{20}
AUIPC	rd=x0	2^{20}
ADDI	rd=x0, and either rs1≠x0 or imm≠0	$2^{17} - 1$
ANDI	rd=x0	2^{17}
ORI	rd=x0	2^{17}
XORI	rd=x0	2^{17}
ADD	rd=x0	2^{10}
SUB	rd=x0	2^{10}
AND	rd=x0	2^{10}
OR	rd=x0	2^{10}
XOR	rd=x0	2^{10}
SLL	rd=x0	2^{10}
SRL	rd=x0	2^{10}
SRA	rd=x0	2^{10}
FENCE	pred=0 or succ=0	$2^{5} - 1$

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.2.177 2.9 (p.28) Table 2.3 R Following HINT instructions are reserved for custom HINTs: no standard HINTs will ever be defined in this subspace.

Instruction	Constraints	Code Points
SLTI	rd=x0	2^{17}
SLTIU	rd=x0	2^{17}
SLLI	rd=x0	2^{10}
SRLI	rd=x0	2^{10}
SRAI	rd=x0	2^{10}
SLT	rd=x0	2^{10}
SLTU	rd=x0	2^{10}

RVU.2.178 2.9 (p.28) C No standard hints are presently defined. We anticipate standard hints to eventually include memory-system spatial and temporal locality hints, branch prediction hints, thread-scheduling hints, security tags, and instrumentation flags for simulation/emulation.

CHAPTER 3 “Zifencei” Instruction-Fetch Fence

ID	REFERENCE	TYPE	DEFINITION
R.3.1	3.0 (p.31)	H	“Zifencei” Instruction-Fetch Fence
R.3.2	3.0 (p.31) preface (p.i)	I	Zifencei extension version is 2.0 and status is ratified.
R.3.3	3.0 (p.31)	I	The “Zifencei” extension includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart. Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetches.
R.3.4	3.0 (p.31)	C	We considered but did not include a “store instruction word” instruction (as in MAJC). 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.
R.3.5	3.0 (p.31)	C	The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, or if the memory system consists of only uncached RAMs, then just the fetch pipeline needs to be flushed at a FENCE.I.
R.3.6	3.0 (p.31)	C	The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.
R.3.7	3.0 (p.31)	C	First, it has been recognized that on some systems, FENCE.I will be expensive to implement and alternate mechanisms are being discussed in the memory model task group. In particular, for designs that have an incoherent instruction cache and an incoherent data cache, or where the instruction cache refill does not snoop a coherent data cache, both caches must be completely flushed when a FENCE.I instruction is encountered. This problem is exacerbated when there are multiple levels of I and D cache in front of a unified cache or outer memory system.
R.3.8	3.0 (p.31, p.32)	C	Second, the instruction is not powerful enough to make available at user level in a Unix-like operating system environment. The FENCE.I only synchronizes the local hart, and the OS can reschedule the user hart to a different physical hart after the FENCE.I. This would require the OS to execute an additional FENCE.I as part of every context migration. For this reason, the standard Linux ABI has removed FENCE.I from user-level and now requires a system call to maintain instruction-fetch coherence, which allows the OS to minimize the number of FENCE.I executions required on current systems and provides forward-compatibility with future improved instruction-fetch coherence mechanisms.

ID	REFERENCE	TYPE	DEFINITION
R.3.9	3.0 (p.32)	C	Future approaches to instruction-fetch coherence under discussion include providing more restricted versions of FENCE.I that only target a given address specified in rs1, and/or allowing software to use an ABI that relies on machine-mode cache-maintenance operations.
R.3.10	3.0 (p.32) & 24.0 (p.129, p.131)	R	FENCE.I Instruction Synchronize the instruction and data streams **Encoding**: I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[11:0]												rs1					FENCE.I			rd					MISC_MEM						
																	0	0	1			0	0	0	0	1	1	1	1		

Valid Base: RV32, RV64, RV128
Task: –
Explanation: The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction. A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does not ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.
Special Case: none
Exception: none

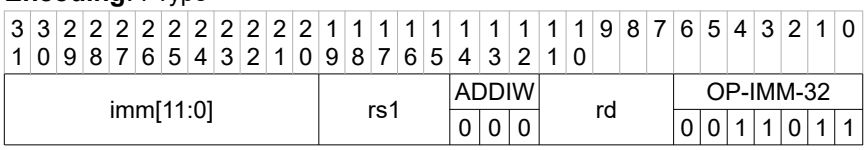
R.3.11	3.0 (p.32)	R	The unused fields in the FENCE.I instruction, *imm[11:0]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.
R.3.12	3.0 (p.32)	C	Because FENCE.I only orders stores with a hart's own instruction fetches, application code should only rely upon FENCE.I if the application thread will not be migrated to a different hart. The EEI can provide mechanisms for efficient multiprocessor instruction-stream synchronization.

CHAPTER 4 RV32E Base Integer Instruction Set

ID	REFERENCE	TYPE	DEFINITION
RVU.4.1	4.0 (p.33)	H	RV32E Base Integer Instruction Set
RVU.4.2	4.0 (p.33) preface (p.i)	I	RV32E version is 1.9 and status is draft.
RVU.4.3	4.0 (p.33)	I	RV32E base integer instruction set is a reduced version of RV32I designed for embedded systems.
RVU.4.4	4.0 (p.33)	C	RV32E was designed to provide an even smaller base core for embedded microcontrollers. Although we had mentioned this possibility in version 2.0 of this document, we initially resisted defining this subset. However, given the demand for the smallest possible 32-bit microcontroller, and in the interests of preempting fragmentation in this space, we have now defined RV32E as a fourth standard base ISA in addition to RV32I, RV64I, and RV128I. There is also interest in defining an RV64E to reduce context state for highly threaded 64-bit processors.
RVU.4.5	4.1 (p.33)	H	RV32E Programmers' Model
RVU.4.6	4.1 (p.33)	T	RV32E reduces the integer register count to 16 general-purpose registers, (x0–x15), where x0 is a dedicated zero register.
RVU.4.7	4.1 (p.33)	C	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.
RVU.4.8	4.1 (p.33)	C	This change requires a different calling convention and ABI. In particular, RV32E is only used with a soft-float calling convention. A new embedded ABI is under consideration that would work across RV32E and RV32I.
RVU.4.9	4.2 (p.34)	H	RV32E Instruction Set
RVU.4.10	4.2 (p.34)	T	RV32E uses the same instruction-set encoding as RV32I, except that only registers x0–x15 are provided.
RVU.4.11	4.2 (p.34)	T	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 custom extensions.
RVU.4.12	4.2 (p.34)	C	RV32E can be combined with all current standard extensions. Defining the F, D, and Q extensions as having a 16-entry floating point register file when combined with RV32E was considered but decided against. To support systems with reduced floating-point register state, we intend to define a "Zfinx" extension that makes floating-point computations use the integer registers, removing the floating-point loads, stores, and moves between floating point and integer registers.

CHAPTER 5 RV64I Base Integer Instruction Set

ID	REFERENCE	TYPE	DEFINITION
RVU.5.1	5.0 (p.35)	H	RV64I Base Integer Instruction Set
RVU.5.2	5.0 (p.35) preface (p.i)	I	RV64I version is 2.1 and status is ratified.
RVU.5.3	5.0 (p.35)	R	RV64I base integer instruction set builds upon the RV32I variant.
RVU.5.4	5.0 (p.35)	I	This chapter presents only the differences with RV32I, so should be read in conjunction with the earlier chapter.
RVU.5.5	5.1 (p.35)	H	Register State
RVU.5.6	5.1 (p.35)	R	RV64I widens the integer registers to 64 bits (XLEN=64) (Number of registers is still 32 as in RV32I)
RVU.5.7	5.1 (p.35)	R	RV64I widens the supported user address space to 64 bits
RVU.5.8	5.2 (p.35)	H	Integer Computational Instructions
RVU.5.9	5.2 (p.35)	I	Most integer computational instructions operate on XLEN-bit values. Additional instruction variants are provided to manipulate 32-bit values in RV64I, indicated by a 'W' suffix to the opcode.
RVU.5.10	5.2 (p.35)	R	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.
RVU.5.11	5.2 (p.35)	C	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.
RVU.5.12	5.2 (p.36)	H	Integer Register-Immediate Instructions
RVU.5.13	5.2 (p.36) & 24.0 (p.129, p.131)	R	ADDIW Instruction 32 bit Integer register-immediate signed addition operation **Encoding**: I-Type



Valid Base: RV64, RV128
Task: `sum32=x(rs1)[31:0]+I-Immediate[31:0]`
`x(rd)[31:0]=sum32; x(rd)[63:32]=sum32[31]`
Explanation: ADDIW 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.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.14	5.2 (p.36)	I	Note, `ADDIW rd, rs1, 0` writes the sign-extension of the lower 32 bits of register *rs1* into register *rd* (assembler pseudoinstruction `SEXT.W`).
RVU.5.15	5.2 (p.36) & 24.0 (p.129, p.131)	R	SLLI Instruction Logical left shift register by immediate operation **Encoding**: Special I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	imm[5:0]						rs1					SLLI			rd					OP-IMM						
																	0	0	1						0	0	1	0	0	1	1

Valid Base: RV64
Task: `x(rd)=x(rs1)<<I-Immediate[5:0]`
Explanation: SLLI is a logical left shift operation (zeros are shifted into the lower bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 63).
Higher order bits of I-Immediate (I-Immediate[11:6]) are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.16	5.2 (p.36) & 24.0 (p.129, p.131)	R	SRLI Instruction Logical right shift register by immediate operation **Encoding**: Special I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	imm[5:0]						rs1					SRLI/ SRAI			rd					OP-IMM						
																	1	0	1						0	0	1	0	0	1	1

Valid Base: RV64
Task: `x(rd)=x(rs1)>>I-Immediate[5:0]`
Explanation: SRLI is a logical right shift operation (zeros are shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 63).
The value of I-Immediate[10] bit determines if this is a logical or an arithmetic shift. For SRLI, I-Immediate[10]=0.
Other higher order bits of I-Immediate are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.17	5.2 (p.36) & 24.0 (p.129, p.131)	R	SRAI: Arithmetic right shift register by immediate operation **Encoding**: Special I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	imm[5:0]						rs1					SRLI/ SRAI			rd			OP-IMM								
																	1	0	1				0	0	1	0	0	1	1		

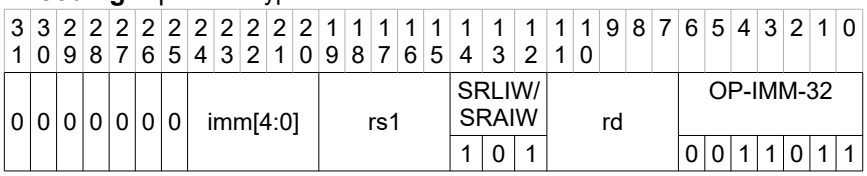
Valid Base: RV64
Task: `x(rd)=x(rs1)>>>I-Immediate[5:0]`
Explanation: SRAI is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 63).
The value of I-Immediate[10] bit determines if this is a logical or arithmetic shift. For SRAI, I-Immediate[10]=1.
Other higher order bits of I-Immediate are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.18	5.2 (p.36) & 24.0 (p.129, p.131)	R	SLLIW Instruction 32 bit logical left shift register by immediate operation **Encoding**: Special I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	imm[4:0]					rs1					SLLIW			rd			OP-IMM-32								
																	0	0	1				0	0	1	1	0	1	1		

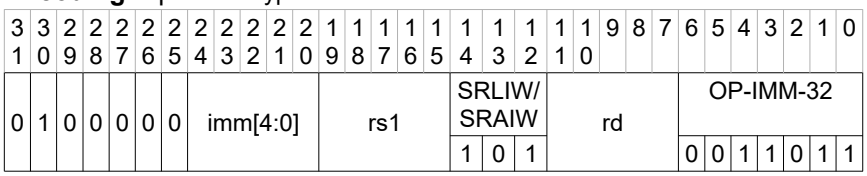
Valid Base: RV64, RV128
Task:
```
shft32=x(rs1)[31:0]<<I-Immediate[4:0]
x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]
```
Explanation: SLLIW is a logical left shift operation (zeros are shifted into the lower bits) applied to the lower 32 bits of *rs1* register and the result is stored into *rd* register as sign extended XLEN bit number. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31).
Higher order bits of I-Immediate (I-Immediate[11:5]) are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.19	5.2 (p.36) & 24.0 (p.129, p.131)	R	SRLIW Instruction 32 bit logical right shift register by immediate operation **Encoding**: Special I-Type



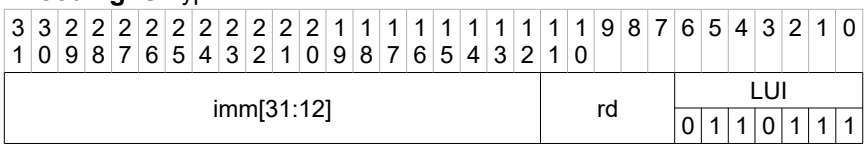
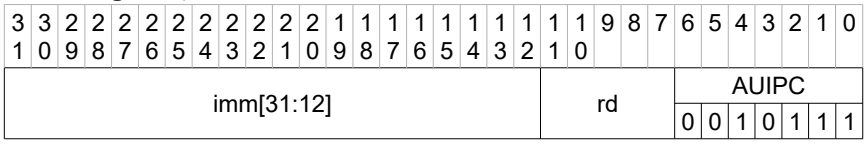
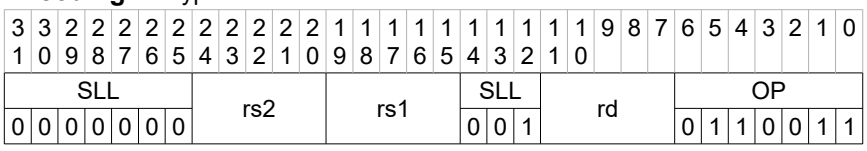
Valid Base: RV64, RV128
Task: `shft32=x(rs1)[31:0]>>I-Immediate[4:0]`
`x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]`
Explanation: SRLIW is a logical right shift operation (zeros are shifted into the higher bits) applied to the lower 32 bits of the *rs1* register and the result is stored into *rd* register as sign extended XLEN bit number. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31).
The value of I-Immediate[10] bit determines if this is a logical or arithmetic shift. For SRLIW, I-Immediate[10]=0.
Other higher order bits of I-Immediate (I-Immediate[11] & I-Immediate[9:5]) are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.20	5.2 (p.36) & 24.0 (p.129, p.131)	R	SRAIW Instruction 32 bit arithmetic right shift register by immediate operation **Encoding**: Special I-Type

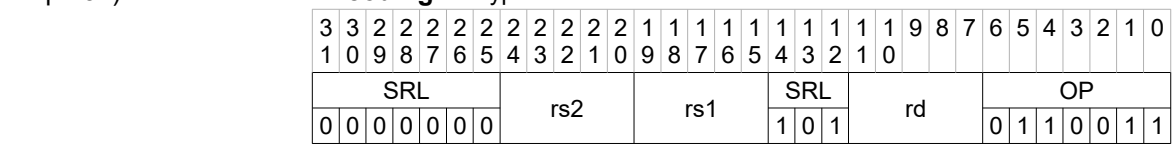


Valid Base: RV64, RV128
Task: `shft32=x(rs1)[31:0]>>>I-Immediate[4:0]`
`x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]`
Explanation: SRAIW is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the lower 32 bits of the *rs1* register and the result is stored into *rd* register as sign extended XLEN number. First 5 bits of I-Immediate (treated as an unsigned integer) determines the amount of shift (0 to 31).
The value of I-Immediate[10] bit determines if this is a logical or arithmetic shift. For SRAI, I-Immediate[10]=1.
Other higher order bits of I-Immediate (I-Immediate[11] & I-Immediate[9:5]) are expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.21	5.2 (p.36)	R	SLLIW, SRLIW, and SRAIW instructions with imm[5] ≠ 0 are reserved
RVU.5.22	5.2 (p.36)	C	Previously, SLLIW, SRLIW, and SRAIW with imm[5] ≠ 0 were defined to cause illegal instruction exceptions, whereas now they are marked as reserved. This is a backwards-compatiblechange.

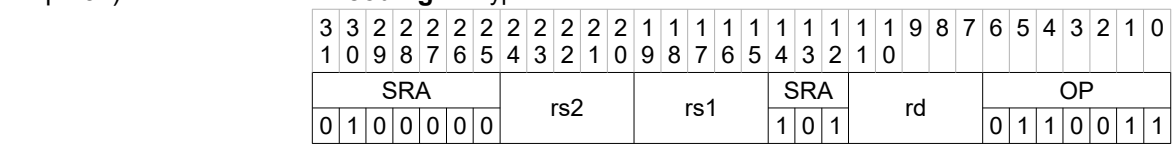
ID	REFERENCE	TYPE	DEFINITION
RVU.5.23	5.2 (p.36) & 24.0 (p.129, p.131)	R	LUI Instruction Load upper immediate operation **Encoding**: U-Type  **Valid Base**: RV64 **Task**: `x(rd)=U-Immediate[63:0]` **Explanation**: LUI places the 20-bit U-immediate into bits 31–12 of register rd and places zero in the lowest 12 bits. The 32-bit result is sign-extended to 64 bits. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.5.24	5.2 (p.37) & 24.0 (p.129, p.131)	R	AUIPC Instruction Add upper immediate to PC operation **Encoding**: U-Type  **Valid Base**: RV64 **Task**: `x(rd)=PC+U-Immediate[63:0]` **Explanation**: AUIPC 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, adds it to the address of the AUIPC instruction, then places the result in register *rd*. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.5.25	5.2 (p.37)	H	Integer Register-Register Operations
RVU.5.26	5.2 (p.37) & 24.0 (p.129, p.131)	R	SLL Instruction Logical left shift register by register operation **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `x(rd)=x(rs1)<<x(rs2)[5:0]` **Explanation**: SLL is a logical left shift operation (zeros are shifted into the lower bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 63). **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.27	5.2 (p.37) & 24.0 (p.129, p.131)	R	SRL Instruction Logical right shift register by register operation **Encoding**: R-Type



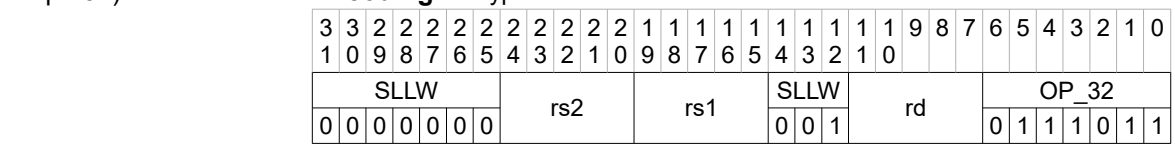
Valid Base: RV64
Task: `x(rd)=x(rs1)>>x(rs2)[5:0]`
Explanation: SRL is a logical right shift operation (zeros are shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 63).
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.28	5.2 (p.37) & 24.0 (p.129, p.131)	R	SRA Instruction Arithmetic right shift register by register operation **Encoding**: R-Type



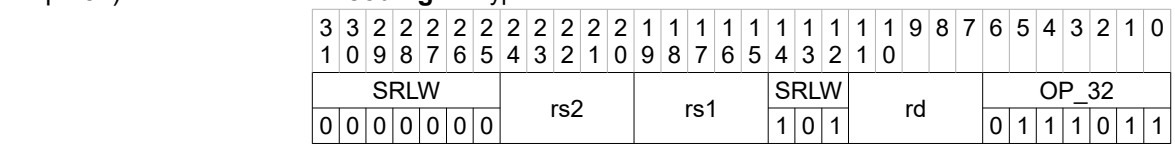
Valid Base: RV32
Task: `x(rd)=x(rs1)>>>x(rs2)[5:0]`
Explanation: SRA is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the contents of the *rs1* register and the result is stored into *rd* register. First 6 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 63).
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.5.29	5.2 (p.37) & 24.0 (p.129, p.131)	R	SLLW Instruction 32 bit Logical left shift register by register operation **Encoding**: R-Type



Valid Base: RV64, RV128
Task:
```
shft32=x(rs1)[31:0]<<x(rs2)[4:0]
x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]
```
Explanation: SLLW is a logical left shift operation (zeros are shifted into the lower bits) applied to the low 32 bits of the *rs1* register and the result is stored into *rd* register as sign extended. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31).
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.30	5.2 (p.37) & 24.0 (p.129, p.131)	R	SRLW Instruction 32 bit Logical right shift register by register operation **Encoding**: R-Type



Valid Base: RV64, RV128

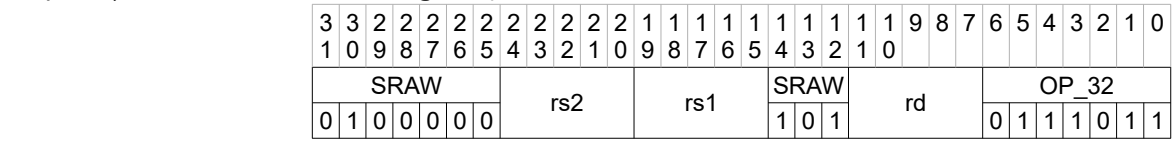
Task:
```
shft32=x(rs1)[31:0]>>x(rs2)[4:0]
x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]
```

Explanation: SRLW is a logical right shift operation (zeros are shifted into the higher bits) applied to the low 32 bits of the *rs1* register and the result is stored into *rd* register as sign extended. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31).

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Exception: none

RVU.5.31	5.2 (p.37) & 24.0 (p.129, p.131)	R	SRAW Instruction 32 bit Arithmetic right shift register by register operation **Encoding**: R-Type



Valid Base: RV64, RV128

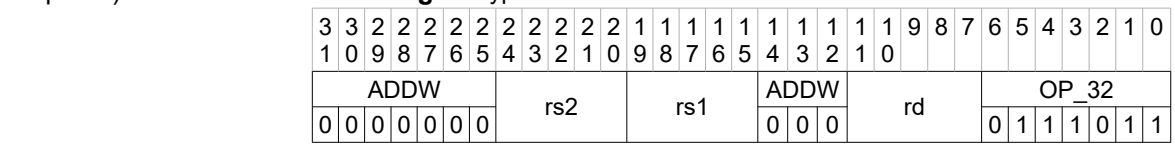
Task:
```
shft32=x(rs1)[31:0]>>>x(rs2)[4:0]
x(rd)[31:0]=shft32; x(rd)[63:32]=shft32[31]
```

Explanation: SRAW is an arithmetic right shift operation (sign bit is shifted into the higher bits) applied to the low 32 bits of the *rs1* register and the result is stored into *rd* register as sign extended. First 5 bits of register *rs2* (treated as an unsigned integer) determines the amount of shift (0 to 31).

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Exception: none

RVU.5.32	5.2 (p.37) & 24.0 (p.129, p.131)	R	ADDW Instruction 32 bit Integer register- register signed addition operation **Encoding**: R-Type



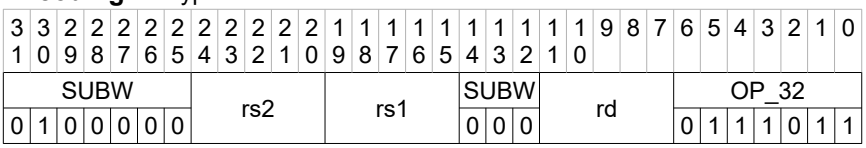
Valid Base: RV64, RV128

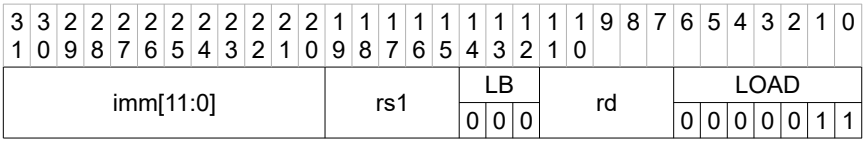
Task:
```
sum32=x(rs1)[31:0]+x(rs2)[31:0]
x(rd)[31:0]=sum32; x(rd)[63:32]=sum32[31]
```

Explanation: ADDW is a summation operation applied to the low 32 bits of the *rs1* & *rs2* registers and the result is stored into *rd* register as sign extended. Overflows are ignored.

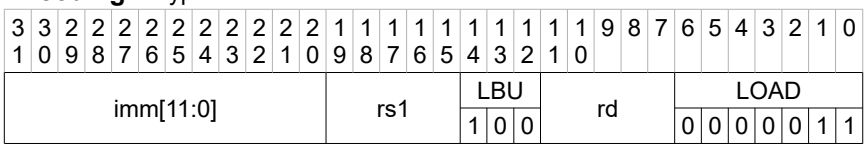
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

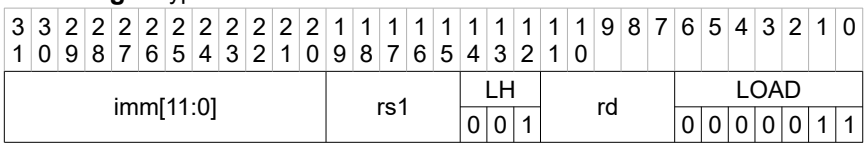
Exception: none

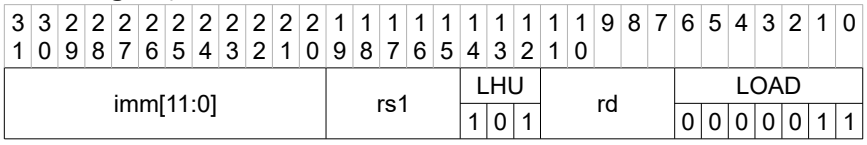
ID	REFERENCE	TYPE	DEFINITION
RVU.5.33	5.2 (p.37) & 24.0 (p.129, p.131)	R	SUBW Instruction 32 bit Integer register- register signed subtraction operation **Encoding**: R-Type  **Valid Base**: RV64, RV128 **Task**: `sub32=x(rs1)[31:0]-x(rs2)[31:0]` `x(rd)[31:0]=sub32; x(rd)[63:32]=sub32[31]` **Explanation**: SUBW is a subtraction operation applied to the low 32 bits of the *rs1* & *rs2* registers and the result is stored into *rd* register as sign extended. Overflows are ignored. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.5.34	5.3 (p.37)	H	Load and Store Instructions
RVU.5.35	5.3 (p.37)	R	RV64I extends the address space to 64 bits.
RVU.5.36	5.3 (p.37)	I	The execution environment will define what portions of the address space are legal to access.

RVU.5.37	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LB Instruction
			Load signed byte from memory to register operation
			Encoding: I-Type
			 
			
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr)`
			`x(rd)[7:0]=mem_val; x(rd)[63:8]=mem_val[7]`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one byte (8-bit) value from memory at the effective address, then sign-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

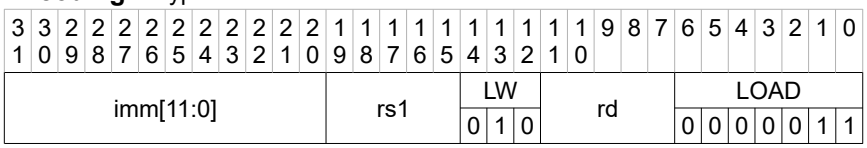
ID	REFERENCE	TYPE	DEFINITION

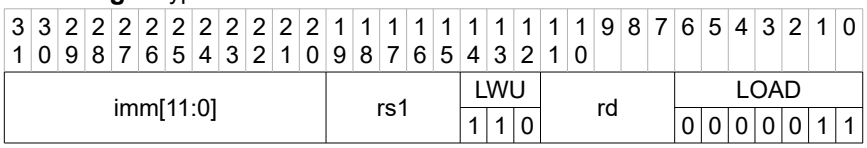
RVU.5.38	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LBU Instruction
			Load unsigned byte from memory to register operation
			Encoding: I-Type
			 
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr)`
			`x(rd)[7:0]=mem_val; x(rd)[63:8]=0`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one byte (8-bit) value from memory at the effective address, then zero-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

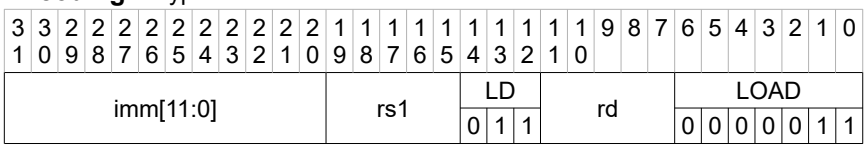
RVU.5.39	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LH Instruction
			Load signed 2 bytes from memory to register operation
			Encoding: I-Type
			 
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+1)`
			`x(rd)[15:0]=mem_val; x(rd)[63:16]=mem_val[15]`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 2 bytes (16-bit) value from memory at the effective address, then sign-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

RVU.5.40	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LHU Instruction
			Load unsigned 2 bytes from memory to register operation
			Encoding: I-Type
			 
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+1)`
			`x(rd)[15:0]=mem_val; x(rd)[63:16]=0`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 2 bytes (16-bit) value from memory at the effective address, then zero-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

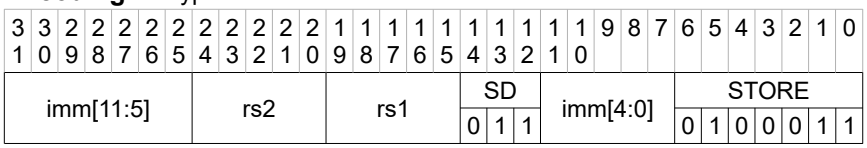
ID	REFERENCE	TYPE	DEFINITION

RVU.5.41	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LW Instruction
			Load signed 4 bytes from memory to register operation
			Encoding: I-Type
			 
			
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+3)`
			`x(rd)[31:0]=mem_val; x(rd)[63:32]=mem_val[31]`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 4 bytes (32-bit) value from memory at the effective address, then sign-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

RVU.5.42	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LWU Instruction
			Load unsigned 4 bytes from memory to register operation
			Encoding: I-Type
			 
			
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+3)`
			`x(rd)[31:0]=mem_val; x(rd)[63:32]=0`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 4 bytes (32-bit) value from memory at the effective address, then zero-extends to 64-bits and stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

RVU.5.43	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	LD Instruction
			Load 8 bytes from memory to register operation
			Encoding: I-Type
			 
			
			Valid Base: RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+7)`
			`x(rd)=mem_val`
			Explanation: The memory address is obtained by adding register *rs1* to the I-Immediate value. It loads one 8 bytes (64-bit) value from memory at the effective address, then stores it into *rd*.
			Special Case: If *rd* is zero (addressing x0) mem_val is not saved but read is done
			Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.5.44	5.3 (p.37, p.38) & 24.0 (p.129, p.131)	R	SD Instruction Store 8 bytes from register to memory operation **Encoding**: S-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
imm[11:5]							rs2					rs1					SD			imm[4:0]							STORE						
																	0	1	1								0	1	0	0	0	1	1

Valid Base: RV64

Task:
```
mem_addr=x(rs1)+S-Immediate
MEM(mem_addr:mem_addr+7)=x(rs2)[63:0]
```

Explanation: The memory address is obtained by adding register *rs1* to the S-Immediate value. It stores the contents of *rs2* in to the memory at the effective address.
Special Case: none
Exception: none

RVU.5.45	5.4 (p.38)	H	HINT Instructions
RVU.5.46	5.4 (p.38)	R	All instructions that are microarchitectural HINTs in RV32I are also HINTs in RV64I.
RVU.5.47	5.4 (p.38)	I	The additional computational instructions in RV64I expand both the standard and custom HINT encoding spaces.
RVU.5.48	5.4 (p.38) & 24.0 (p.129, p.131)	R	Following additional HINT instructions are reserved for standard HINTs in addition to RV32I standard HINTs. But none are presently defined.

Instruction	Constraints	Code Points
ADDIW	rd=x0	2^{17}
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}

RVU.5.49	5.4 (p.38) & 24.0 (p.129, p.131)	R	Following additional HINT instructions are reserved for custom HINTs in addition to RV32I custom HINTs. no standard HINTs will ever be defined in this subspace.

Instruction	Constraints	Code Points
SLLIW	rd=x0	2^{10}
SRLIW	rd=x0	2^{10}
SRAIW	rd=x0	2^{10}

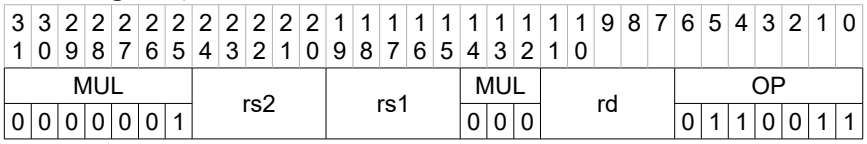
CHAPTER 6 RV128I Base Integer Instruction Set

ID	REFERENCE	TYPE	DEFINITION
RVU.6.1	6.0 (p.41)	H	RV128I Base Integer Instruction Set
RVU.6.2	6.0 (p.41) preface (p.i)	I	RV128I extension version is 1.7 and status is draft.
RVU.6.3	6.0 (p.41)	I	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.
RVU.6.4	6.0 (p.41)	C	The primary reason to extend integer register width is to support larger address spaces. It is not clear when a flat address space larger than 64 bits will be required. At the time of writing, the fastest supercomputer in the world as measured by the Top500 benchmark had over 1PB of DRAM, and would require over 50 bits of address space if all the DRAM resided in a single address space. Some warehouse-scale computers already contain even larger quantities of DRAM, and new dense solid-state non-volatile memories and fast interconnect technologies might drive a demand for even larger memory spaces. Exascale systems research is targeting 100PB memory systems, which occupy 57 bits of address space. At historic rates of growth, it is possible that greater than 64 bits of address space might be required before 2030. History suggests that whenever it becomes clear that more than 64 bits of address space is needed, architects will repeat intensive debates about alternatives to extending the address space, including segmentation, 96-bit address spaces, and software workarounds, until, finally, flat 128-bit address spaces will be adopted as the simplest and best solution. We have not frozen the RV128 spec at this time, as there might be need to evolve the design based on actual usage of 128-bit address spaces.
RVU.6.5	6.0 (p.41)	T	RV128I builds upon RV64I in the same way RV64I builds upon RV32I, with integer registers extended to 128 bits (i.e., XLEN=128).
RVU.6.6	6.0 (p.41)	I	Most integer computational instructions are unchanged as they are defined to operate on XLEN bits.
RVU.6.7	6.0 (p.41)	T	The RV64I “*W” integer instructions that operate on 32-bit values in the low bits of a register are retained but now sign extend their results from bit 31 to bit 127.
RVU.6.8	6.0 (p.41)	T	A new set of “*D” integer instructions are added that operate on 64-bit values held in the low bits of the 128-bit integer registers and sign extend their results from bit 63 to bit 127.
RVU.6.9	6.0 (p.41)	T	The “*D” instructions consume two major opcodes (OP-IMM-64 and OP-64) in the standard 32-bit encoding.
RVU.6.10	6.0 (p.42)	C	To improve compatibility with RV64, in a reverse of how RV32 to RV64 was handled, we might change the decoding around to rename RV64I ADD as a 64-bit ADDD, and add a 128-bit ADDQ in what was previously the OP-64 major opcode (now renamed the OP-128 major opcode).
RVU.6.11	6.0 (p.42)	T	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.6.12	6.0 (p.42)	T	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.
RVU.6.13	6.0 (p.42)	T	SQ is added to the STORE major opcode, while LQ is added to the MISC-MEM major opcode.
RVU.6.14	6.0 (p.42)	T	The floating-point instruction set is unchanged, although the 128-bit Q floating-point extension can now support FMV.X.Q and FMV.Q.X instructions, together with additional FCVT instructions to and from the T (128-bit) integer format.

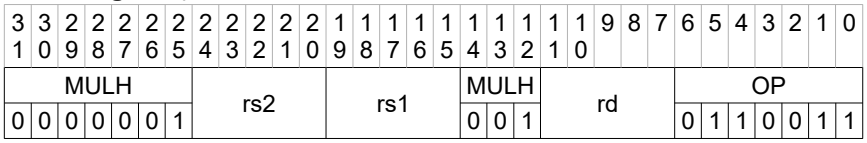
CHAPTER 7 "M" Standard Extension for Integer Multiplication and Division

ID	REFERENCE	TYPE	DEFINITION
RVU.7.1	7.0 (p.43)	H	"M" Standard Extension for Integer Multiplication and Division
RVU.7.2	7.0 (p.43) preface (p.i)	I	"M" extension version is 2.0 and status is ratified.
RVU.7.3	7.0 (p.43)	I	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.
RVU.7.4	7.0 (p.43)	C	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
RVU.7.5	7.1 (p.43)	H	Multiplication Operations
RVU.7.6	7.1 (p.43) & 24.0 (p.129, p.131)	R	MUL Instruction Integer register-register multiplication operation **Encoding**: R-Type



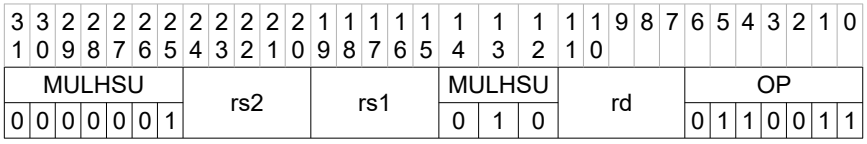
Valid Base: RV32, RV64, RV128
Task: `mult=x(rs1)*x(rs2)`
`x(rd)=mult[XLEN-1:0]`
Explanation: MUL performs an XLEN-bit×XLEN-bit multiplication of *rs1* by *rs2* and places the lower XLEN bits into the destination register.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.7.7	7.1 (p.43) & 24.0 (p.129, p.131)	R	MULH Instruction Integer register-register signed multiplication high word operation **Encoding**: R-Type



Valid Base: RV32, RV64, RV128
Task: `mult=x(rs1)*x(rs2)`
`x(rd)=mult[2*XLEN-1:XLEN]`
Explanation: MULH performs an XLEN-bit×XLEN-bit signedxsigned multiplication of *rs1* by *rs2* and places the higher XLEN bits of the result into the destination register.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.7.8	7.1 (p.43) & 24.0 (p.129, p.131)	R	MULHSU Instruction Integer register-register signed-unsigned multiplication high word operation **Encoding**: R-Type

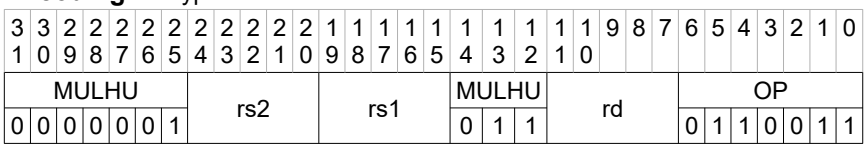


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULHSU							rs2					rs1					MULHSU			rd							OP					
0	0	0	0	0	0	1											0	1	0							0	1	1	0	0	1	1

| RVU.7.8 | 7.1 (p.43) & 24.0 (p.129, p.131) | R | MULHSU Instruction
Integer register-register signed-unsigned multiplication high word operation
Encoding: R-Type |

Valid Base: RV32, RV64, RV128
Task: `mult=x(rs1)*x(rs2)`
`x(rd)=mult[2*XLEN-1:XLEN]`
Explanation: MULHSU performs an XLEN-bit×XLEN-bit signed(*rs1*) x unsigned(*rs2*) multiplication of *rs1* by *rs2* and places the higher XLEN bits of the result into the destination register.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

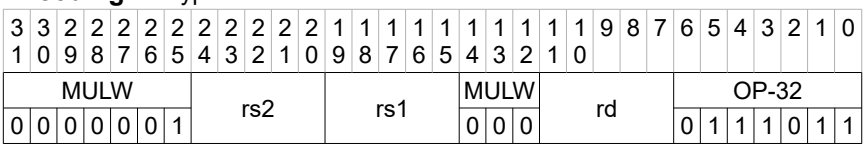
RVU.7.9	7.1 (p.43) & 24.0 (p.129, p.131)	R	MULHU Instruction Integer register-register unsigned multiplication high word operation **Encoding**: R-Type

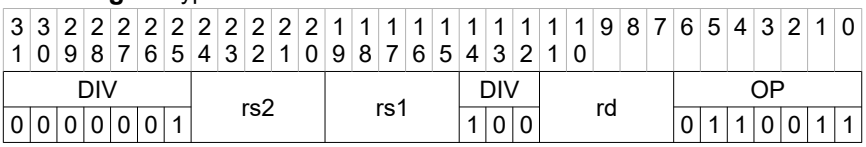


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULHU							rs2					rs1					MULHU			rd							OP					
0	0	0	0	0	0	1											0	1	1							0	1	1	0	0	1	1

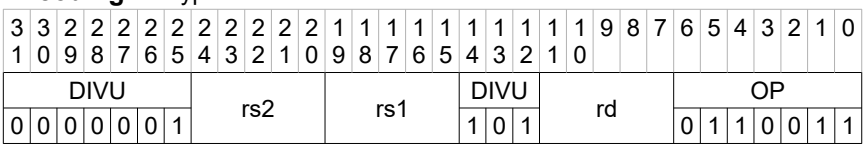
Valid Base: RV32, RV64, RV128
Task: `mult=x(rs1)*x(rs2)`
`x(rd)=mult[2*XLEN-1:XLEN]`
Explanation: MULHU performs an XLEN-bit×XLEN-bit unsigned x unsigned multiplication of *rs1* by *rs2* and places the higher XLEN bits of the result into the destination register.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.7.10	7.1 (p.43)	I	If both the high and low bits of the same product are required, then the recommended code sequence is: `MULH[W][S]U rdh, rs1, rs2; MUL[W] rdl, rs1, rs2` (source register specifiers must be in same order and *rdh* cannot be the same as *rs1* or *rs2*).
RVU.7.11	7.1 (p.43)	O	Microarchitectures can then fuse these (MULHX and MULX) into a single multiply operation instead of performing two separate multiplies.

ID	REFERENCE	TYPE	DEFINITION
RVU.7.12	7.1 (p.44) & 24.0 (p.129, p.131)	R	MULW Instruction 32 bit Integer register-register multiplication operation **Encoding**: R-Type  **Valid Base**: RV64, RV128 **Task**: `mult=x(rs1)[31:0]*x(rs2)[31:0]` `x(rd)[31:0]=mult[31:0]; x(rd)[63:32]=mult[31]` **Explanation**: MULW performs an 32×32 multiplication of *rs1* by *rs2* and places the lower 32 bits into the destination register by sign extending to XLEN bits. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.7.13	7.1 (p.44)	C	MULHSU is used in multi-word signed multiplication to multiply the most-significant word of the multiplicand (which contains the sign bit) with the less-significant words of the multiplier (which are unsigned).
RVU.7.14	7.1 (p.44)	C	In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].
RVU.7.15	7.2 (p.44)	H	Division Operations

RVU.7.16	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	DIV Instruction
			Integer register-register signed division operation
			Encoding: R-Type
			 
			Valid Base: RV32, RV64, RV128
			Task: `(quotient,remainder)=x(rs1)/x(rs2)`
			`where x(rs1)=x(rs2)*quotient+remainder`
			`x(rd)=quotient`
			Explanation: DIV perform an XLEN bits by XLEN bits signed integer division of *rs1* by *rs2*, rounding towards zero. It stores quotient into *rd* as XLEN bit signed number.
			Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
			Divide by zero: When x(*rs2*) is zero, quotient will be set to -1 (all 1s).
			Overflow: When x(*rs1*)=-2^{L-1} and x(*rs2*)=-1, quotient will be set to -2^{L-1}.
			Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.7.17	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	DIVU Instruction Integer register-register unsigned division operation **Encoding**: R-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIVU							rs2					rs1				DIVU			rd					OP							
0	0	0	0	0	0	1										1	0	1				0	1	1	0	0	1	1			

Valid Base: RV32, RV64, RV128

Task:
```
(quotient,remainder)=x(rs1)/x(rs2)
where x(rs1)=x(rs2)*quotient+remainder
x(rd)=quotient
```

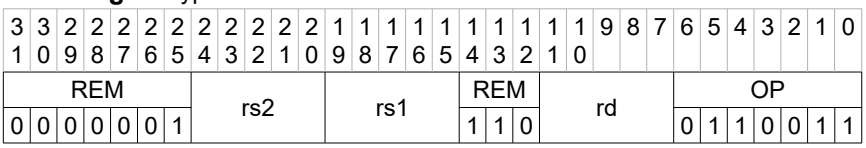
Explanation: DIVU perform an XLEN bits by XLEN bits unsigned integer division of *rs1* by *rs2*, rounding towards zero. It stores quotient into *rd* as XLEN bit unsigned number.

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Divide by zero: When x(*rs2*) is zero, quotient will be set to $2^{L}-1$.

Exception: none

RVU.7.18	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	REM Instruction Integer register-register signed division remainder operation **Encoding**: R-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
REM							rs2					rs1				REM			rd					OP							
0	0	0	0	0	0	1										1	1	0				0	1	1	0	0	1	1			

Valid Base: RV32, RV64, RV128

Task:
```
(quotient,remainder)=x(rs1)/x(rs2)
where x(rs1)=x(rs2)*quotient+remainder
x(rd)=remainder
```

Explanation: REM perform an XLEN bits by XLEN bits signed integer division of *rs1* by *rs2*, rounding towards zero. It stores remainder into *rd* as XLEN bit unsigned number. tThe sign of the result equals the sign of the dividend, x(*rs1*).

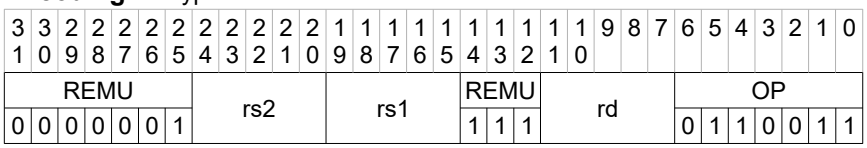
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Divide by zero: When x(*rs2*) is zero, remainder will be set to x(*rs1*).

Overflow: When x(*rs1*)=-2^{L-1} and x(*rs2*)=-1, remainder will be set to 0.

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.7.19	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	REMU Instruction Integer register-register unsigned division remainder operation **Encoding**: R-Type



Valid Base: RV32, RV64, RV128

Task:
```
(quotient,remainder)=x(rs1)/x(rs2)
where x(rs1)=x(rs2)*quotient+remainder
x(rd)=remainder
```

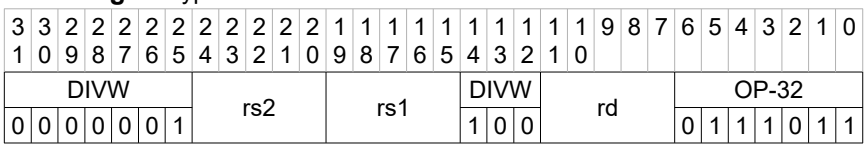
Explanation: REMU perform an XLEN bits by XLEN bits unsigned integer division of *rs1* by *rs2*, rounding towards zero. It stores remainder into *rd* as XLEN bit unsigned number.

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Divide by zero: When x(*rs2*) is zero, remainder will be set to x(*rs1*).

Exception: none

RVU.7.20	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	DIVW Instruction 32-bit integer register-register signed division operation **Encoding**: R-Type



Valid Base: RV64, RV128

Task:
```
(quotient,remainder)=x(rs1)[31:0]/x(rs2)[31:0]
where x(rs1)[31:0]=x(rs2)[31:0]*quotient+remainder
x(rd)[31:0]=quotient; x(rd)[63:32]=quotient[31]
```

Explanation: DIVW performs an 32×32 bit signed division of *rs1* by *rs2* and places the 32 bits quotient into the destination register by sign extending to XLEN bits.

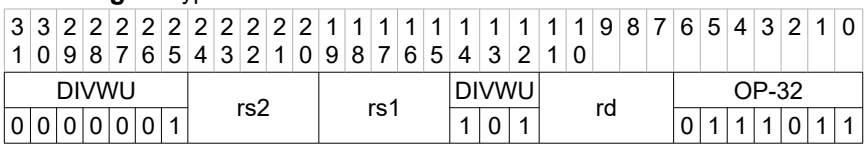
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Divide by zero: When x(*rs2*) is zero, quotient will be set to -1 (all 1s).

Overflow: When x(*rs1*)=-2^{L-1} and x(*rs2*)=-1, quotient will be set to -2^{L-1}.

Exception: none

RVU.7.21	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	DIVWU Instruction 32-bit integer register-register unsigned division operation **Encoding**: R-Type



Valid Base: RV64, RV128

Task:
```
(quotient,remainder)=x(rs1)[31:0]/x(rs2)[31:0]
where x(rs1)[31:0]=x(rs2)[31:0]*quotient+remainder
x(rd)[31:0]=quotient; x(rd)[63:32]=0
```

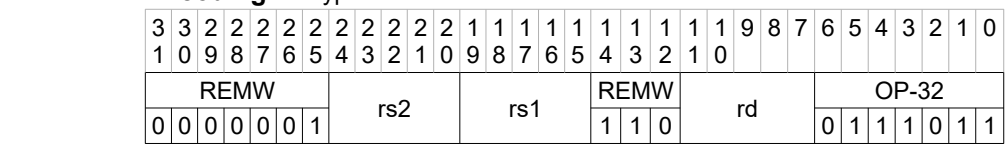
Explanation: DIVWU performs an 32×32 bit unsigned division of rs1 by rs2 and places the 32 bits quotient into the destination register by zero extending to XLEN bits.

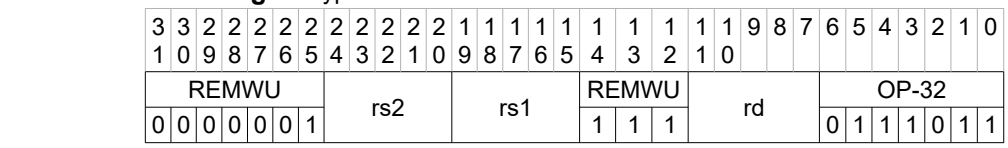
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Divide by zero: When x(*rs2*) is zero, quotient will be set to 2^{L}-1.

Exception: none

ID	REFERENCE	TYPE	DEFINITION

RVU.7.22	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	REMW Instruction
			32-bit integer register-register signed division remainder operation
			Encoding: R-Type
			 
			
			Valid Base: RV64, RV128
			Task: `(quotient,remainder)=x(rs1)[31:0]/x(rs2)[31:0]`
			`where x(rs1)[31:0]=x(rs2)[31:0]*quotient+remainder`
			`x(rd)[31:0]=remainder; x(rd)[63:32]=remainder[31]`
			Explanation: REMW performs an 32×32 bit signed division of *rs1* by *rs2* and places the 32 bits remainder into the destination register by sign extending to XLEN bits
			Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
			Divide by zero: When x(*rs2*) is zero, remainder will be set to x(*rs1*).
			Overflow: When x(*rs1*)=-2^{L-1} and x(*rs2*)=-1, remainder will be set to 0.
			Exception: none

RVU.7.23	7.2 (p.44) & 24.0 (p.129, p.131) Table 7.1	R	REMWU Instruction
			32-bit integer register-register unsigned division operation
			Encoding: R-Type
			 
			
			Valid Base: RV64, RV128
			Task: `(quotient,remainder)=x(rs1)[31:0]/x(rs2)[31:0]`
			`where x(rs1)[31:0]=x(rs2)[31:0]*quotient+remainder`
			`x(rd)[31:0]=remainder; x(rd)[63:32]=0`
			Explanation: REMWU performs an 32×32 bit unsigned division of *rs1* by *rs2* and places the 32 bits remainder into the destination register by zero extending to XLEN bits.
			Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
			Divide by zero: When x(*rs2*) is zero, remainder will be set to x(*rs1*).
			Exception: none

RVU.7.24	7.2 (p.44)	C	For both signed and unsigned division, it holds that `dividend = divisor × quotient + remainder`.
RVU.7.25	7.2 (p.44)	I	If both the quotient and remainder are required from the same division, the recommended code sequence is: `DIV[W][U] rdq, rs1, rs2; REM[W][U] rdr, rs1, rs2` (`rdq` cannot be the same as `rs1` or `rs2`).
RVU.7.26	7.2 (p.44)	O	Microarchitectures can then fuse these (DIVX and REMX) into a single divide operation instead of performing two separate divides.
RVU.7.27	7.2 (p.44) Table 7.1	R	The quotient of division by zero has all bits set, and the remainder of division by zero equals the dividend.
RVU.7.28	7.2 (p.44) Table 7.1	R	Signed division overflow occurs only when the most-negative integer (-2^{L-1}) is divided by −1 (only for DIV[W] & REM[W]). The quotient of a signed division with overflow is equal to the dividend, and the remainder is zero.

ID	REFERENCE	TYPE	DEFINITION
RVU.7.29	7.2 (p.44, p.45)	C	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.
RVU.7.30	7.2 (p.45)	C	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

ID	REFERENCE	TYPE	DEFINITION
RVU.8.1	8.0 (p.47)	H	"A" Standard Extension for Atomic Instructions
RVU.8.2	8.0 (p.47) preface (p.i)	I	"A" extension version is 2.1 and status is ratified.
RVU.8.3	8.0 (p.47)	I	The standard atomic-instruction extension, named "A", contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space.
RVU.8.4	8.0 (p.47)	I	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**
RVU.8.5	8.0 (p.47)	C	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.
RVU.8.6	8.1 (p.47)	H	Specifying Ordering of Atomic Instructions
RVU.8.7	8.1 (p.47)	I	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.
RVU.8.8	8.1 (p.47)	R	To provide more efficient support for release consistency, 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.
RVU.8.9	8.1 (p.47)	R	No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains.
RVU.8.10	8.1 (p.47)	R	If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation.
RVU.8.11	8.1 (p.47)	R	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.
RVU.8.12	8.1 (p.47, p.48)	R	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.
RVU.8.13	8.1 (p.48)	R	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.
RVU.8.14	8.2 (p.48)	H	Load-Reserved/Store-Conditional Instructions

**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.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.15	8.2 (p.48)	I	Complex atomic memory operations on a single memory word or doubleword are performed with the load-reserved (LR) and store-conditional (SC) instructions.
RVU.8.16	8.2 (p.48) & 24.0 (p.129, p.132)	R	LR.W Instruction Load word and reserve operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LR.W					aq	rl	rs2					rs1					LR.W			rd					AMO							
0	0	0	1	0													0	1	0						0	1	0	1	1	1	1

Valid Base: RV32, RV64, RV128

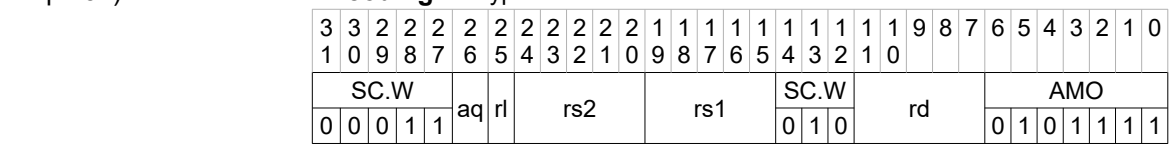
Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val;
x(rd)[XLEN-1:32]=mem_val[31]
RESERVED=[mem_addr:mem_addr+3]
```

Explanation: LR.W loads a word from the address in *rs1*, places the sign-extended value in *rd*, and registers a *reservation set*—a set of bytes that subsumes the bytes in the addressed word. The field *rs2* is not used and expected to be 0.

Special Case: If rd is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.17	8.2 (p.48) & 24.0 (p.129, p.132)	R	SC.W Instruction Store word conditionally operation **Encoding**: R-Type



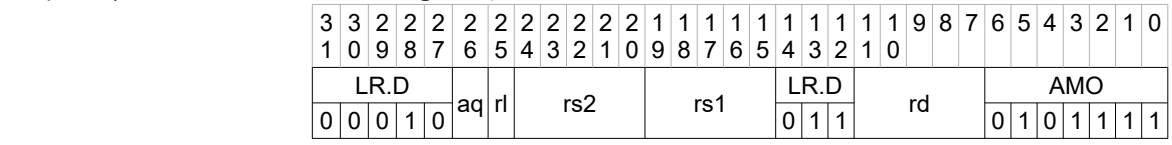
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SC.W					aq	rl	rs2					rs1					SC.W			rd			AMO									
0	0	0	1	1													0	1	0				0	1	0	1	1	1	1		

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
if RESERVED valid
      MEM(mem_addr:mem_addr+3)=x(rs2)[31:0]
      x(rd)=0
else
      x(rd)=non-zero
end
RESERVED=invalid
```

Explanation: SC.W conditionally writes a word in *rs2* to the address in *rs1*: the SC.W succeeds only if the reservation is still valid and the reservation set contains the bytes being written.
If the SC.W succeeds, the instruction writes the word in *rs2* to memory, and it writes zero to *rd*.
If the SC.W fails, the instruction does not write to memory, and it writes a nonzero value to *rd*.
Regardless of success or failure, executing an SC.W instruction invalidates any reservation held by this hart.
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.
Special Case: If rd is zero (addressing x0), return value is ignored but operation is completed
Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

RVU.8.18	8.2 (p.48) & 24.0 (p.129, p.132)	R	LR.D Instruction Load double word and reserve operation **Encoding**: R-Type



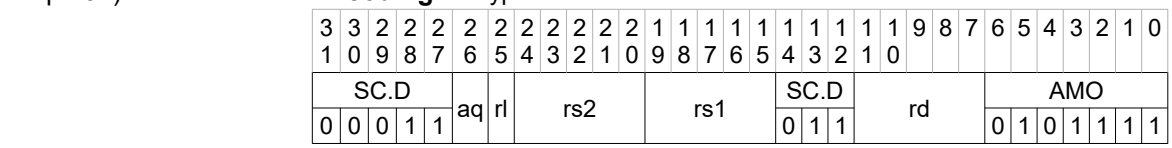
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LR.D					aq	rl	rs2					rs1					LR.D			rd			AMO									
0	0	0	1	0													0	1	1				0	1	0	1	1	1	1		

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val;
x(rd)[XLEN-1:64]=mem_val[63]
RESERVED=[mem_addr:mem_addr+7]
```

Explanation: LR.D loads a double word from the address in *rs1*, places the sign-extended value in *rd*, and registers a reservation set—a set of bytes that subsumes the bytes in the addressed word. The field *rs2* is not used and expected to be 0.
Special Case: If rd is zero (addressing x0), return value is ignored but operation is completed
Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION

RVU.8.19	8.2 (p.48) & 24.0 (p.129, p.132)	R	SC.D Instruction
			Store double word conditionally operation
			Encoding: R-Type
			 
			Valid Base: RV64, RV128
			Task:
			`mem_addr=x(rs1)`
			`if RESERVED valid`
			`  MEM(mem_addr:mem_addr+7)=x(rs2)[63:0]`
			`  x(rd)=0`
			`else`
			`  x(rd)=non-zero`
			`end`
			`RESERVED=invalid`
			Explanation: SC.D conditionally writes a double word in *rs2* to the address in *rs1* as described in SC.W instruction.
			Special Case: If rd is zero (addressing x0), return value is ignored but operation is completed
			Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

RVU.8.20	8.2 (p.49)	O	The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.
RVU.8.21	8.2 (p.49)	C	Emulating misaligned LR/SC sequences is impractical in most systems. Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.
RVU.8.22	8.2 (p.49)	C	(in SC.W & SC.D) 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.
RVU.8.23	8.2 (p.48)	C	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).

ID	REFERENCE	TYPE	DEFINITION
RVU.8.24	8.2 (p.48)	C	The main disadvantage of LR/SC over CAS is livelock, which we avoid, under certain circumstances, with an architected guarantee of eventual forward progress as described below. Another concern is whether the influence of the current x86 architecture, with its DW-CAS, will complicate porting of synchronization libraries and other software that assumes DW-CAS is the basic machine primitive. A possible mitigating factor is the recent addition of transactional memory instructions to x86, which might cause a move away from DW-CAS.
RVU.8.25	8.2 (p.48, p.49)	C	More generally, a multi-word atomic primitive is desirable, but there is still considerable debate about what form this should take, and guaranteeing forward progress adds complexity to a system. Our current thoughts are to include a small limited-capacity transactional memory buffer along the lines of the original transactional memory proposals as an optional standard extension “T”.
RVU.8.26	8.2 (p.49)	R	An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or double word.
RVU.8.27	8.2 (p.49)	R	An SC can only pair with the most recent LR in program order.
RVU.8.28	8.2 (p.49)	R	An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order.
RVU.8.29	8.2 (p.49)	R	An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC’s address as part of the reservation set.
RVU.8.30	8.2 (p.49)	C	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.
RVU.8.31	8.2 (p.49)	C	To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only required to invalidate reservations when they overlap the bytes accessed by the LR. These writes are not required to invalidate the reservation when they access other bytes in the reservation set.
RVU.8.32	8.2 (p.49)	R	The SC must fail if the address is not within the reservation set of the most recent LR in program order.
RVU.8.33	8.2 (p.49)	R	The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC.
RVU.8.34	8.2 (p.49)	R	The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.)
RVU.8.35	8.2 (p.49)	R	An SC must fail if there is another SC (to any address) between the LR and the SC in program order.
RVU.8.36	8.2 (p.49)	I	The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom in Section 14.1 “RVWMO Memory Consistency Model”
RVU.8.37	8.2 (p.49)	C	The platform should provide a means to determine the size and shape of the reservation set.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.38	8.2 (p.50)	C	A platform specification may constrain the size and shape of the reservation set. For example, the Unix platform is expected to require of main memory that the reservation set be of fixed size, contiguous, naturally aligned, and no greater than the virtual memory page size.
RVU.8.39	8.2 (p.50)	C	A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation: • during a preemptive context switch, and • if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.
RVU.8.40	8.2 (p.50)	C	The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 14.1 that ensures software runs correctly on expected common implementations that operate in this manner.
RVU.8.41	8.2 (p.50)	R	An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.
RVU.8.42	8.2 (p.50)	I	The LR/SC sequence can be given acquire semantics by setting the *aq* bit on the LR instruction.
RVU.8.43	8.2 (p.50)	I	The LR/SC sequence can be given release semantics by setting the *rl* bit on the SC instruction.
RVU.8.44	8.2 (p.50)	R	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.
RVU.8.45	8.2 (p.50)	R	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.
RVU.8.46	8.2 (p.50)	R	Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set, nor should software set the *aq* bit on an SC instruction unless the *rl* bit is also set.
RVU.8.47	8.2 (p.50)	I	LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

ID	REFERENCE	TYPE	DEFINITION

RVU.8.48	8.2 (p.51) Figure 8.1	I	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 below. If inlined, compare-and-swap functionality need only take four instructions.
			Sample code for compare-and-swap function using LR/SC
			`# a0 holds address of memory location`
			`# a1 holds expected value`
			`# a2 holds desired value`
			`# a0 holds return value,0 if successful,!0 otherwise`
			`cas:`
			`lr.w t0, (a0) # Load original value.`
			`bne t0, a1, fail # Doesn't match, so fail.`
			`sc.w t0, a2, (a0) # Try to update.`
			`bnez t0, cas # Retry if store-conditional failed.`
			`li a0, 0 # Set return to success.`
			`jr ra # Return.`
			`fail:`
			`li a0, 1 # Set return to failure.`
			`jr ra # Return.`

RVU.8.49	8.3 (p.51)	H	Eventual Success of Store-Conditional Instructions
RVU.8.50	8.3 (p.51)	R	The standard A extension defines constrained LR/SC loops, which have the following properties: • The loop comprises only an LR/SC sequence and code to retry the sequence in the case of failure, and must comprise at most 16 instructions placed sequentially in memory. • An LR/SC sequence begins with an LR instruction and ends with an SC instruction. The dynamic code executed between the LR and SC instructions can only contain instructions from the base "I" instruction set, excluding loads, stores, backward jumps, taken backward branches, JALR, FENCE, FENCE.I, and SYSTEM instructions. If the "C" extension is supported, then compressed forms of the aforementioned "I" instructions are also permitted. • The code to retry a failing LR/SC sequence can contain backwards jumps and/or branches to repeat the LR/SC sequence, but otherwise has the same constraint as the code between the LR and SC. • The LR and SC addresses must lie within a memory region with the LR/SC eventuality property. The execution environment is responsible for communicating which regions have this property. • The SC must be to the same effective address and of the same data size as the latest LR executed by the same hart.
RVU.8.51	8.3 (p.51)	I	LR/SC sequences that do not lie within constrained LR/SC loops are *unconstrained*.
RVU.8.52	8.3 (p.51)	O	Unconstrained LR/SC sequences might succeed on some attempts on some implementations, but might never succeed on other implementations.
RVU.8.53	8.3 (p.51)	C	We restricted the length of LR/SC loops to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and associativity. Similarly, we disallowed other loads and stores within the loops to avoid restrictions on data-cache associativity in simple implementations that track the reservation within a private cache. The restrictions on branches and jumps limit the time that can be spent in the sequence. Floatingpoint operations and integer multiply/divide were disallowed to simplify the operating system's emulation of these instructions on implementations lacking appropriate hardware support.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.54	8.3 (p.51)	C	Software is not forbidden from using unconstrained LR/SC sequences, but portable software must detect the case that the sequence repeatedly fails, then fall back to an alternate code sequence that does not rely on an unconstrained LR/SC sequence. Implementations are permitted to unconditionally fail any unconstrained LR/SC sequence.
RVU.8.55	8.3 (p.51, p.52)	R	If a hart H enters a constrained LR/SC loop, the execution environment must guarantee that one of the following events eventually occurs: • H or some other hart executes a successful SC to the reservation set of the LR instruction in H's constrained LR/SC loops. • Some other hart executes an unconditional store or AMO instruction to the reservation set of the LR instruction in H's constrained LR/SC loop, or some other device in the system writes to that reservation set. • H executes a branch or jump that exits the constrained LR/SC loop. • H traps.
RVU.8.56	8.3 (p.52)	C	Note that these definitions permit an implementation to fail an SC instruction occasionally for any reason, provided the aforementioned guarantee is not violated.
RVU.8.57	8.3 (p.52)	C	As a consequence of the eventuality guarantee, if some harts in an execution environment are executing constrained LR/SC loops, and no other harts or devices in the execution environment execute an unconditional store or AMO to that reservation set, then at least one hart will eventually exit its constrained LR/SC loop. By contrast, if other harts or devices continue to write to that reservation set, it is not guaranteed that any hart will exit its LR/SC loop.
RVU.8.58	8.3 (p.52)	C	Loads and load-reserved instructions do not by themselves impede the progress of other harts' LR/SC sequences. We note this constraint implies, among other things, that loads and loadreserved instructions executed by other harts (possibly within the same core) cannot impede LR/SC progress indefinitely. For example, cache evictions caused by another hart sharing the cache cannot impede LR/SC progress indefinitely. Typically, this implies reservations are tracked independently of evictions from any shared cache. Similarly, cache misses caused by speculative execution within a hart cannot impede LR/SC progress indefinitely.
RVU.8.59	8.3 (p.52)	C	These definitions admit the possibility that SC instructions may spuriously fail for implementation reasons, provided progress is eventually made.
RVU.8.60	8.3 (p.52)	C	One advantage of CAS is that it guarantees that some hart eventually makes progress, whereas an LR/SC atomic sequence could livelock indefinitely on some systems. To avoid this concern, we added an architectural guarantee of livelock freedom for certain LR/SC sequences.
RVU.8.61	8.3 (p.52)	C	Earlier versions of this specification imposed a stronger starvation-freedom guarantee. However, the weaker livelock-freedom guarantee is sufficient to implement the C11 and C++11 languages, and is substantially easier to provide in some microarchitectural styles.
RVU.8.62	8.4 (p.52)	H	Atomic Memory Operations
RVU.8.63	8.4 (p.52)	I	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*.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.64	8.4 (p.52)	R	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*.
RVU.8.65	8.4 (p.53)	R	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).
RVU.8.66	8.4 (p.53)	R	If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated.
RVU.8.67	8.4 (p.53)	R	The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.
RVU.8.68	8.4 (p.53)	I	The “Zam” extension, described in Chapter 22, relaxes this requirement and specifies the semantics of misaligned AMOs.
RVU.8.69	8.4 (p.53)	I	The operations supported are swap, integer add, bitwise AND, bitwise OR, bitwise XOR, and signed and unsigned integer maximum and minimum.
RVU.8.70	8.4 (p.53)	I	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.
RVU.8.71	8.4 (p.53)	C	We provided fetch-and-op style atomic primitives as they scale to highly parallel systems better than LR/SC or CAS. A simple microarchitecture can implement AMOs using the LR/SC primitives, provided the implementation can guarantee the AMO eventually completes. More complex implementations might also implement AMOs at memory controllers, and can optimize away fetching the original value when the destination is x0.
RVU.8.72	8.4 (p.53)	C	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.
RVU.8.73	8.4 (p.53)	R	To help implement multiprocessor synchronization, the AMOs optionally provide release consistency semantics.
RVU.8.74	8.4 (p.53)	R	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.
RVU.8.75	8.4 (p.53)	R	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.
RVU.8.76	8.4 (p.53)	R	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.
RVU.8.77	8.4 (p.53)	C	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.

ID	REFERENCE	TYPE	DEFINITION

RVU.8.78	8.4 (p.53) Figure 8.2	I	An example code sequence for a critical section guarded by a test-and-test-and-set spinlock is shown below.
			Sample code for mutual exclusion. a0 contains the address of the lock.
			`li t0, 1 # Initialize swap value.`
			`again:`
			`lw t1, (a0) # Check if lock is held.`
			`bnez t1, again # Retry if held.`
			`amoswap.w.aq t1, t0, (a0) # Attempt to acquire lock.`
			`bnez t1, again # Retry if held.`
			`# ...`
			`# Critical section.`
			`# ...`
			`amoswap.w.rl x0, x0, (a0) #Release lock by storing 0`

RVU.8.79	8.4 (p.53)	I	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.
RVU.8.80	8.4 (p.53)	C	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.
RVU.8.81	8.4 (p.53)	I	The instructions in the “A” extension can also be used to provide sequentially consistent loads and stores.
RVU.8.82	8.4 (p.53)	I	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.
RVU.8.83	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOSWAP.W Instruction Atomic read-swap-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOSWAP.W					aq	rl	rs2					rs1					.W			rd					AMO							
0	0	0	0	1													0	1	0						0	1	0	1	1	1	1

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=x(rs2)[31:0]
```

Explanation: AMOSWAP.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then writes the 32 bit content of *rs2* to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.84	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOSWAP.D Instructiopn Atomic read-swap-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOSWAP.D					aq	rl	rs2					rs1					.D			rd			AMO								
0	0	0	0	1													0	1	1				0	1	0	1	1	1	1		

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=x(rs2)[63:0]
```

Explanation: AMOSWAP.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then writes the 64 bit content of *rs2* to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.85	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOADD.W Instruction Atomic read-add-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOADD.W					aq	rl	rs2					rs1					.W			rd			AMO								
0	0	0	0	0													0	1	0				0	1	0	1	1	1	1		

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=mem_val+x(rs2)[31:0]
```

Explanation: AMOADD.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then adds the 32 bit content of *rs2* to the loaded value and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.86	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOADD.D Instruction Atomic read-add-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOADD.D						aq	rl	rs2					rs1				.D			rd							AMO					
0	0	0	0	0													0	1	1								0	1	0	1	1	1	1

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=mem_val+x(rs2)[63:0]
```

Explanation: AMOADD.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then adds the 32 bit content of *rs2* to the loaded value and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.87	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOAND.W Instruction Atomic read-and-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOAND.W						aq	rl	rs2					rs1				.W			rd							AMO					
0	1	1	0	0													0	1	0								0	1	0	1	1	1	1

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=mem_val and x(rs2)[31:0]
```

Explanation: AMOAND.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise ands the 32 bit content of *rs2* to the loaded value and writes it back to the same memory location.

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.88	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOAND.D Instruction Atomic read-and-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOAND.D					aq	rl	rs2					rs1					.D			rd					AMO							
0	1	1	0	0													0	1	1						0	1	0	1	1	1	1

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=mem_val and x(rs2)[63:0]
```

Explanation: AMOAND.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise ands the 64 bit content of *rs2* to the loaded value and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.89	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOOR.W Instruction Atomic read-or-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOOR.W					aq	rl	rs2					rs1					.W			rd					AMO							
0	1	0	0	0													0	1	0						0	1	0	1	1	1	1

Valid Base: RV32, RV64, RV128

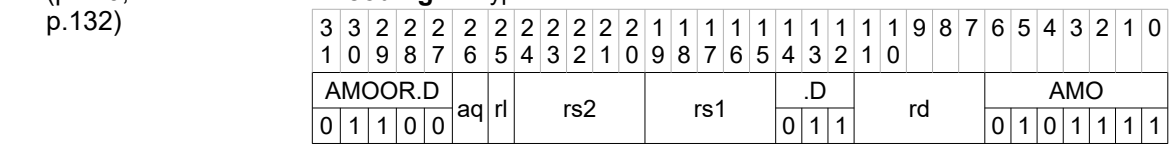
Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=mem_val or x(rs2)[31:0]
```

Explanation: AMOAND.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise ors the 32 bit content of *rs2* to the loaded value and writes it back to the same memory location.

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.90	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOOR.D Instruction Atomic read-or-write memory double word operation **Encoding**: R-Type



Valid Base: RV64, RV128

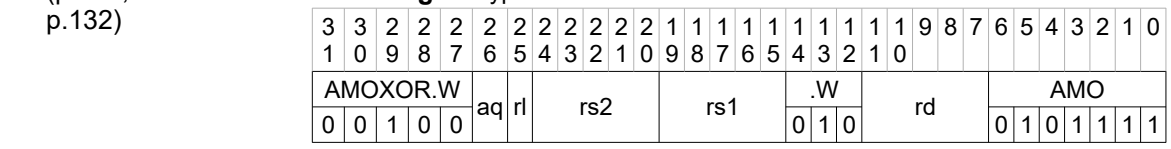
Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=mem_val or x(rs2)[63:0]
```

Explanation: AMOAND.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise ors the 64 bit content of *rs2* to the loaded value and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

RVU.8.91	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOXOR.W Instruction Atomic read-xor-write memory word operation **Encoding**: R-Type



Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=mem_val xor x(rs2)[31:0]
```

Explanation: AMOAND.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise xors the 32 bit content of *rs2* to the loaded value and writes it back to the same memory location.

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.92	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOXOR.D Instruction Atomic read-xor-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOXOR.D					aq	rl	rs2					rs1					.D			rd							AMO					
0	0	1	0	0													0	1	1								0	1	0	1	1	1	1

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=mem_val xor x(rs2)[63:0]
```

Explanation: AMOAND.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then bitwise xors the 64 bit content of *rs2* to the loaded value and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.93	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMIN.W Instruction Atomic read-get signed minimum-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMIN.W					aq	rl	rs2					rs1					.W			rd							AMO					
1	0	0	0	0													0	1	0								0	1	0	1	1	1	1

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=SMIN(mem_val,x(rs2)[31:0])
```

Explanation: AMOMIN.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then finds the signed minimum of loaded value or the 32 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.94	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMIN.D Instruction Atomic read-get signed minimum-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMIN.D					aq	rl	rs2					rs1					.D			rd			AMO								
1	0	0	0	0													0	1	1				0	1	0	1	1	1	1		

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=SMIN(mem_val,x(rs2)[63:0])
```

Explanation: AMOMIN.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then finds the signed minimum of loaded value or the 64 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.95	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMINU.W Instructio Atomic read-get unsigned minimum-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMINU.W					aq	rl	rs2					rs1					.W			rd			AMO								
1	1	0	0	0													0	1	0				0	1	0	1	1	1	1		

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=UMIN(mem_val,x(rs2)[31:0])
```

Explanation: AMOMINU.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then finds the unsigned minimum of loaded value or the 32 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.96	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMINU.D Instruction Atomic read-get unsigned minimum-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMINU.D					aq	rl	rs2					rs1					.D			rd			AMO								
1	0	0	0	0													0	1	1				0	1	0	1	1	1	1		

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=UMIN(mem_val,x(rs2)[63:0])
```

Explanation: AMOMINU.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then finds the unsigned minimum of loaded value or the 64 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.97	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMAX.W Instruction Atomic read-get signed maximum-write memory word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMAX.W					aq	rl	rs2					rs1					.W			rd			AMO								
1	0	1	0	0													0	1	0				0	1	0	1	1	1	1		

Valid Base: RV32, RV64, RV128

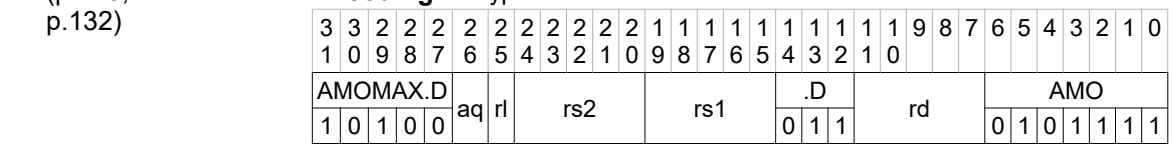
Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=SMAX(mem_val,x(rs2)[31:0])
```

Explanation: AMOMAX.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then finds the signed maximum of loaded value or the 32 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.98	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMAX.D Instruction Atomic read-get signed maximum-write memory double word operation **Encoding**: R-Type



Valid Base: RV64, RV128

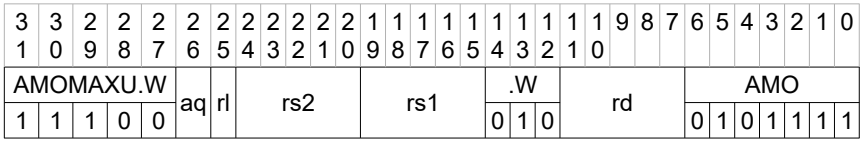
Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=SMAX(mem_val,x(rs2)[63:0])
```

Explanation: AMOMAX.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then finds the signed maximum of loaded value or the 64 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

RVU.8.99	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMAXU.W Instruction Atomic read-get unsigned maximum-write memory word operation **Encoding**: R-Type



Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)[31:0]=mem_val
x(rd)[XLEN-1:32]=mem_val[31]
MEM(mem_addr:mem_addr+3)=UMAX(mem_val,x(rs2)[31:0])
```

Explanation: AMOMAXU.W loads a word (32 bits) from memory address in *rs1* and stores it into *rd*. Then finds the unsigned maximum of loaded value or the 32 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

Exception: If the address is not naturally aligned to 4 bytes, an address misaligned exception or an access-fault exception will be generated.

ID	REFERENCE	TYPE	DEFINITION
RVU.8.100	8.4 (p.52, p.53) & 24.0 (p.129, p.132)	R	AMOMAXU.D Instruction Atomic read-get unsigned maximum-write memory double word operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AMOMAXU.D					aq	rl	rs2					rs1					.D			rd			AMO								
1	1	1	0	0													0	1	1				0	1	0	1	1	1	1		

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1)
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)[63:0]=mem_val
x(rd)[XLEN-1:64]=mem_val[63]
MEM(mem_addr:mem_addr+7)=UMAX(mem_val,x(rs2)[63:0])
```

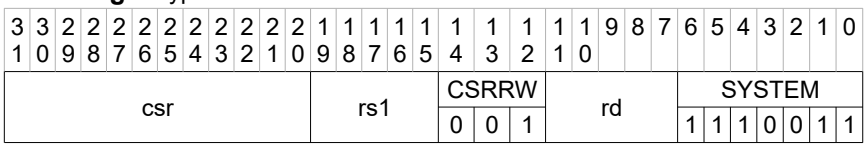
Explanation: AMOMAXU.D loads a double word (64 bits) from memory address in *rs1* and stores it into *rd*. Then finds the unsigned maximum of loaded value or the 64 bit content of *rs2* and writes it back to the same memory location

Special Case: If *rd* is zero (addressing x0), return value is ignored but operation is completed

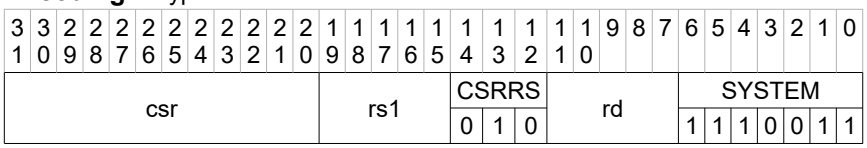
Exception: If the address is not naturally aligned to 8 bytes, an address misaligned exception or an access-fault exception will be generated.

CHAPTER 9 “Zicsr”, Control and Status Register (CSR) Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.9.1	9.0 (p.55)	H	“Zicsr”, Control and Status Register (CSR) Instructions
RVU.9.2	9.0 (p.55) preface (p.i)	I	Zicsr extension version is 2.0 and status is ratified.
RVU.9.3	9.0 (p.55)	R	RISC-V defines a separate address space of 4096 Control and Status registers associated with each hart.
RVU.9.4	9.0 (p.55)	I	This chapter defines the full set of CSR instructions that operate on these CSRs.
RVU.9.5	9.0 (p.55)	C	While CSRs are primarily used by the privileged architecture, there are several uses in unprivileged code including for counters and timers, and for floating-point status.
RVU.9.6	9.0 (p.55)	C	The counters and timers are no longer considered mandatory parts of the standard base ISAs, and so the CSR instructions required to access them have been moved out of the base ISA chapter into this separate chapter.
RVU.9.7	9.1 (p.55)	H	CSR Instructions
RVU.9.8	9.1 (p.55)	R	All CSR instructions atomically read-modify-write a single CSR, ...
RVU.9.9	9.1 (p.55)	R	… whose CSR specifier is encoded in the 12-bit csr field of the instruction held in bits 31–20.
RVU.9.10	9.1 (p.55)	R	The immediate forms use a 5-bit zero-extended immediate encoded in the rs1 field.
RVU.9.11	9.1 (p.55)	R	CSR registers are XLEN bits wide

RVU.9.12	9.1 (p.55) & 24.0 (p.129, p.131) Table 9.1	R	CSRRW Instruction
			Atomic read-write CSR register operation
			Encoding: I-Type
			 
			Valid Base: RV32, RV64, RV128
			Task: `csr_val=CSR(csr)`
			`x(rd)=csr_val`
			`CSR(csr)=x(rs1)`
			Explanation: 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.
			Special Case: If *rd* is zero (addressing 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. Write is done.
			Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.9.13	9.1 (p.56) & 24.0 (p.129, p.131) Table 9.1	R	CSRRS Instruction Atomic read and set bits CSR register operation **Encoding**: I-Type

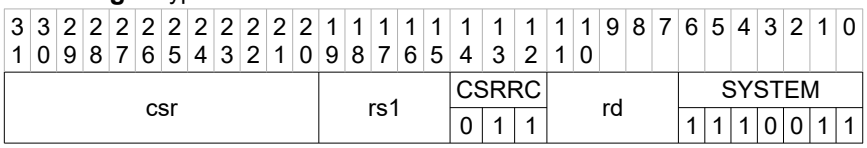


Valid Base: RV32, RV64, RV128

Task:
```
csr_val=CSR(csr)
x(rd)=csr_val
CSR(csr)=csr_val or (csr_write_mask and x(rs1))
```

Explanation: 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).
Special Case: If *rd* is zero (addressing x0), save to *rd* is not done.
if *rs1*=0, 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.
It always reads the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields.
Exception: none

RVU.9.14	9.1 (p.56) & 24.0 (p.129, p.131) Table 9.1	R	CSRRC Instruction Atomic read and clear bits CSR register operation **Encoding**: I-Type



Valid Base: RV32, RV64, RV128

Task:
```
csr_val=CSR(csr)
x(rd)=csr_val
CSR(csr)=csr_val and ~(csr_write_mask and x(rs1))
```

Explanation: 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.
Special Case: If *rd* is zero (addressing x0), save to *rd* is not done.
if *rs1*=0, 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.
It always reads the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.9.15	9.1 (p.56) & 24.0 (p.129, p.131) Table 9.1	R	CSRRWI Instruction Atomic read-write CSR immediate operation **Encoding**: I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
csr												uimm[4:0]					CSRRWI			rd					SYSTEM							
																	1	0	1			1	1	1	0	0	1	1				

Valid Base: RV32, RV64, RV128

Task:
```
csr_val=CSR(csr)
x(rd)=csr_val
CSR(csr)[4:0]=x(rs1);CSR(csr)[XLEN-1:5]=0
```

Explanation: CSRRWI reads the old value of the CSR, zero-extends the value to XLEN bits, then writes it to integer register *rd*. Then zero extended immediate value *uimm* is written to the CSR.

Special Case: If *rd* is zero (addressing 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. Write is done.

Exception: none

RVU.9.16	9.1 (p.56) & 24.0 (p.129, p.131) Table 9.1	R	CSRRSI Instruction Atomic read and set bits CSR immediate operation **Encoding**: I-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
csr												uimm[4:0]					CSRRSI			rd					SYSTEM							
																	1	1	0			1	1	1	0	0	1	1				

Valid Base: RV32, RV64, RV128

Task:
```
csr_val=CSR(csr)
x(rd)=csr_val
zeuimm[4:0]=uimm; zeuimm[XLEN-1:5]=0
CSR(csr)=csr_val or (csr_write_mask and zeuimm)
```

Explanation: The CSRRSI instruction reads the value of the CSR, zero extends the value to XLEN bits, and writes it to integer register *rd*.

The zero extended *uimm* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *uimm* 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).

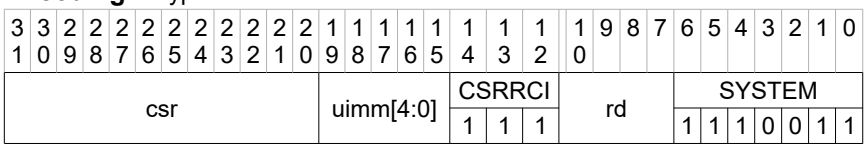
Special Case: If *rd* is zero (addressing x0), save to *rd* is not done.

if *uimm*=0, then the instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.

It always reads the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields.

Exception: none

ID	REFERENCE	TYPE	DEFINITION

RVU.9.17	9.1 (p.56) & 24.0 (p.129, p.131) Table 9.1	R	CSRRCI Instruction
			Atomic read and clear bits CSR immediate operation
			Encoding: I-Type
			 
			Valid Base: RV32, RV64, RV128
			Task: `csr_val=CSR(csr)`
			`x(rd)=csr_val`
			`zeuimm[4:0]=uimm; zeuimm[XLEN-1:5]=0`
			`CSR(csr)=csr_val and ~(csr_write_mask and zeuimm)`
			Explanation: The CSRRCI instruction reads the value of the CSR, zero extends the value to XLEN bits, and writes it to integer register *rd*.
			The zero extended *uimm* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *uimm* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).
			Special Case: If *rd* is zero (addressing x0), save to *rd* is not done.
			if *uimm*=0, then the instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write.
			It always reads the addressed CSR and cause any read side effects regardless of *rs1* and *rd* fields.
			Exception: none

RVU.9.18	9.1 (p.57)	C	The CSRs defined so far do not have any architectural side effects on reads beyond raising illegal instruction exceptions on disallowed accesses. Custom extensions might add CSRs with side effects on reads.
RVU.9.19	9.1 (p.57)	R	Some CSRs, such as the instructions-retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes such a CSR, the write is done instead of the increment. In particular, a value written to instret by one instruction will be the value read by the following instruction.
RVU.9.20	9.1 (p.57)	I	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`.
RVU.9.21	9.1 (p.57)	I	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`.
RVU.9.22	9.1 (p.57)	H	CSR Access Ordering
RVU.9.23	9.1 (p.57)	R	On a given hart, explicit and implicit CSR access are performed in program order with respect to those instructions whose execution behavior is affected by the state of the accessed CSR.
RVU.9.24	9.1 (p.57)	R	In particular, a CSR access is performed after the execution of any prior instructions in program order whose behavior modifies or is modified by the CSR state and before the execution of any subsequent instructions in program order whose behavior modifies or is modified by the CSR state.

ID	REFERENCE	TYPE	DEFINITION
RVU.9.25	9.1 (p.57)	R	Furthermore, a CSR read access instruction returns the accessed CSR state before the execution of the instruction, while a CSR write access instruction updates the accessed CSR state after the execution of the instruction.
RVU.9.26	9.1 (p.57)	R	Where the above program order does not hold, CSR accesses are weakly ordered, and the local hart or other harts may observe the CSR accesses in an order different from program order.
RVU.9.27	9.1 (p.57)	R	In addition, CSR accesses are not ordered with respect to explicit memory accesses, unless a CSR access modifies the execution behavior of the instruction that performs the explicit memory access or unless a CSR access and an explicit memory access are ordered by either the syntactic dependencies defined by the memory model or the ordering requirements defined by the Memory-Ordering PMAs section in Volume II of this manual.
RVU.9.28	9.1 (p.57)	R	To enforce ordering in all other cases, software should execute a FENCE instruction between the relevant accesses.
RVU.9.29	9.1 (p.57)	R	Other than the case of RVU.9.23, CSR accesses are weakly ordered, and the local hart or other harts may observe the CSR accesses in an order different from program order.
RVU.9.30	9.1 (p.57)	R	For the purposes of the FENCE instruction, CSR read accesses are classified as device input (I), and CSR write accesses are classified as device output (O).
RVU.9.31	9.1 (p.57)	C	Informally, the CSR space acts as a weakly ordered memory-mapped I/O region, as defined by the Memory-Ordering PMAs section in Volume II of this manual. As a result, the order of CSR accesses with respect to all other accesses is constrained by the same mechanisms that constrain the order of memory-mapped I/O accesses to such a region.
RVU.9.32	9.1 (p.57, p.58)	C	These CSR-ordering constraints are imposed primarily to support ordering main memory and memory-mapped I/O accesses with respect to reads of the time CSR. With the exception of the *time*, *cycle*, and *mcycle* CSRs, the CSRs defined thus far in Volumes I and II of this specification are not directly accessible to other harts or devices and cause no side effects visible to other harts or devices. Thus, accesses to CSRs other than the aforementioned three can be freely reordered with respect to FENCE instructions without violating this specification.
RVU.9.33	9.1 (p.58)	R	For CSR accesses that cause side effects, the above ordering constraints apply to the order of the initiation of those side effects but does not necessarily apply to the order of the completion of those side effects.
RVU.9.34	9.1 (p.58)	O	The hardware platform may define that accesses to certain CSRs are strongly ordered, as defined by the Memory-Ordering PMAs section in Volume II of this manual. Accesses to strongly ordered CSRs have stronger ordering constraints with respect to accesses to both weakly ordered CSRs and accesses to memory-mapped I/O regions.

CHAPTER 10 Counters

ID	REFERENCE	TYPE	DEFINITION
RVU.10.1	10.0 (p.59)	H	Counters
RVU.10.2	10.0 (p.59) preface (p.i)	I	Counters extension version is 2.0 and status is draft.
RVU.10.3	10.0 (p.59)	R	RISC-V ISAs provide a set of up to 32 performance counters and timers
RVU.10.4	10.0 (p.59)	R	Counters and timers are of 64 bits length
RVU.10.5	10.0 (p.59)	R	When XLEN >= 64 (RV64, RV128), counters and timers are accessed at CSR register address 0xC00–0xC1F
RVU.10.6	10.0 (p.59)	R	When XLEN == 32 (RV32) lower 32 bits of the counters and timers are accessed at CSR register address 0xC00–0xC1F
RVU.10.7	10.0 (p.59)	R	When XLEN == 32 (RV32) higher 32 bits of the counters and timers are accessed at CSR register address 0xC80–0xC9F
RVU.10.8	10.0 (p.59)	R	Counters and timers are read only CSR registers
RVU.10.9	10.0 (p.59)	R	The first three of these (CYCLE, TIME, and INSTRET) have dedicated functions (cycle count, real-time clock, and instructions-retired respectively), ...
RVU.10.10	10.0 (p.59)	O	… while the remaining counters, if implemented, provide programmable event counting.
RVU.10.11	10.1 (p.59)	H	Base Counters and Timers
RVU.10.12	10.1 (p.59)	R	Counters and timers can be read with CSRRS instruction having *rs1* equal to zero and *csr* field (immediate value) equal to the CSR register address
RVU.10.13	10.1 (p.59)	R	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.
RVU.10.14	10.1 (p.59)	R	In RV64I, the CSR instructions can manipulate 64-bit CSRs.
RVU.10.15	10.1 (p.59)	C	Some execution environments might prohibit access to counters to impede timing side-channel attacks.
RVU.10.16	10.0 (p.59) 24.0 (p.136)	R	The first counter at the CSR address 0xC00 (plus 0xC80 for RV32), is named `cycle` and works as cycle counter.
RVU.10.17	10.1 (p.59)	R	The `cycle` counter 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.
RVU.10.18	10.1 (p.59)	R	The underlying 64-bit counter should never overflow in practice.
RVU.10.19	10.1 (p.59)	R	The rate at which the cycle counter advances will depend on the implementation and operating environment.
RVU.10.20	10.1 (p.59)	R	The execution environment should provide a means to determine the current rate (cycles/second) at which the cycle counter is incrementing.

ID	REFERENCE	TYPE	DEFINITION
RVU.10.21	10.1 (p.60)	C	`cycle` 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 `cycle` 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.
RVU.10.22	10.1 (p.60)	C	Cores don’t have to be exposed to software at all, and an implementor might choose to pretend multiple harts on one physical core are running on separate cores with one hart/core, and provide separate cycle counters for each hart. This might make sense in a simple barrel processor (e.g., CDC 6600 peripheral processors) where inter-hart timing interactions are nonexistent or minimal.
RVU.10.23	10.1 (p.60)	C	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.
RVU.10.24	10.1 (p.60)	C	Standardizing what happens during “sleep” is not practical given that what “sleep” means is not standardized across execution environments, but if the entire core is paused (entirely clockgated or powered-down in deep sleep), then it is not executing clock cycles, and the cycle count shouldn’t be increasing per the spec. There are many details, e.g., whether clock cycles required to reset a processor after waking up from a power-down event should be counted, and these are considered execution-environment-specific details.
RVU.10.25	10.1 (p.60)	C	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 `cycle` was primarily performance monitoring/tuning, and the specification was written with that goal in mind.
RVU.10.26	10.0 (p.59) 24.0 (p.136)	R	The second counter at the CSR address 0xC01 (plus 0xC81 for RV32), is named `time` and works as real time clock
RVU.10.27	10.1 (p.60)	R	The `time` counts wall-clock real time that has passed from an arbitrary start time in the past.
RVU.10.28	10.1 (p.60)	R	The underlying 64-bit counter should never overflow in practice.
RVU.10.29	10.1 (p.60)	R	The execution environment should provide a means of determining the period of the real-time counter (seconds/tick). The period must be constant.
RVU.10.30	10.1 (p.60)	R	The real-time clocks of all harts in a single user application should be synchronized to within one tick of the real-time clock.

ID	REFERENCE	TYPE	DEFINITION
RVU.10.31	10.1 (p.60)	R	The environment should provide a means to determine the accuracy of the clock.
RVU.10.32	10.1 (p.60)	R	On some simple platforms, cycle count might represent a valid implementation of `time`, but in this case, platforms should implement the `time` as an alias for `cycle` to make code more portable, rather than using `cycle` to measure wall-clock time.
RVU.10.33	10.0 (p.59) 24.0 (p.136)	R	The third counter at the CSR address 0xC02 (plus 0xC82 for RV32), is named `instret` and works as instructions-retired
RVU.10.34	10.1 (p.60)	R	The `instret` counts the number of instructions retired by this hart from some arbitrary start point in the past.
RVU.10.35	10.1 (p.60)	R	The underlying 64-bit counter should never overflow in practice.

RVU.10.36	10.1 (p.61)	I	The following code sequence will read a valid 64-bit cycle counter value into x3:x2, even if the counter overflows its lower half between reading its upper and lower halves.
			Sample code for reading the 64-bit cycle counter in RV32
			`again:`
			`  csrrs x3, cycleh, x0`
			`  csrrs x2, cycle, x0`
			`  csrrs x4, cycleh, x0`
			`  bne x3, x4, again`

RVU.10.37	10.1 (p.61)	C	We recommend provision of these basic counters in implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.
RVU.10.38	10.1 (p.61)	C	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.
RVU.10.39	10.1 (p.61)	C	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.
RVU.10.40	10.2(p.61)	H	Hardware Performance Counters
RVU.10.41	10.2(p.61)	R	There is CSR space allocated for 29 additional unprivileged 64-bit hardware performance counters, `hpmcounter3`–`hpmcounter31`.
RVU.10.42	10.2(p.61)	R	For RV32, the upper 32 bits of these performance counters is accessible via additional CSRs `hpmcounter3h`–`hpmcounter31h`.
RVU.10.43	10.2(p.61)	R	These counters count platform-specific events and are configured via additional privileged registers. The number and width of these additional counters, and the set of events they count is platform-specific.

ID	REFERENCE	TYPE	DEFINITION
RVU.10.44	10.2(p.61)	C	The privileged architecture manual describes the privileged CSRs controlling access to these counters and to set the events to be counted.
RVU.10.45	10.2(p.61)	C	It would be useful to eventually standardize event settings to count ISA-level metrics, such as the number of floating-point instructions executed for example, and possibly a few common microarchitectural metrics, such as “L1 instruction cache misses”.

CHAPTER 11 “F” Standard Extension for Single-Precision Floating-Point

ID	REFERENCE	TYPE	DEFINITION
RVU.11.1	11.0 (p.63)	H	“F” Standard Extension for Single-Precision Floating-Point
RVU.11.2	11.0 (p.63) preface (p.i)	I	“F” extension version is 2.2 and status is ratified.
RVU.11.3	11.0 (p.63)	I	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††
RVU.11.4	11.0 (p.63)	R	The F extension depends on the “Zicsr” extension for control and status register access.
RVU.11.5	11.1 (p.63)	H	F Register State
RVU.11.6	11.1 (p.63)	R	The F extension adds 32 floating-point registers, f0–f31, each 32 bits wide.
RVU.11.7	11.1 (p.63)	R	The F extension adds a floating-point control and status register `fcsr`, which is 32 bits wide and contains the operating mode and exception status of the floating-point unit.
RVU.11.8	11.2 (p.65)	R	`fcsr` is a read-write RISC-V control and status register (CSR).
RVU.11.9	24.0 (p.136)	R	CSR address of `fcsr` is 0x0003
RVU.11.10	11.1 (p.63)	I	We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, ...
RVU.11.11	11.1 (p.63)	R	… and FLEN=32 for the F single-precision floating-point extension.
RVU.11.12	11.1 (p.63)	I	Most floating-point instructions operate on values in the floating-point register file.
RVU.11.13	11.1 (p.63)	I	Floating-point load and store instructions transfer floating-point values between registers and memory.
RVU.11.14	11.1 (p.63)	I	Instructions to transfer values to and from the integer register file are also provided.
RVU.11.15	11.1 (p.63)	C	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.
RVU.11.16	11.2 (p.65)	H	Floating-Point Control and Status Register
RVU.11.17	11.2 (p.65)	R	`fcsr` bit mapping

31	...	8	7	6	5	4	3	2	1	0
Reserved			Rounding Mode (frm)			Acrued Exceptions (fflags)				
						NV	DZ	OF	UF	NX

†† ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008

ID	REFERENCE	TYPE	DEFINITION
RVU.11.18	11.2 (p.65)	R	The fields within the `fcsr` can also be accessed individually through different CSR addresses.
RVU.11.19	24.0 (p.136)	R	CSR address of *frm* (Floating-Point Dynamic Rounding Mode) is 0x0002
RVU.11.20	24.0 (p.136)	R	CSR address of *fflags* (Floating-Point Accrued Exceptions.) is 0x0001
RVU.11.21	11.2 (p.65)	R	Bits 31–8 of the fcsr are reserved for other standard extensions, including the “L” standard extension for decimal floating-point.
RVU.11.22	11.2 (p.65)	R	If these extensions (that makes use of bits 31-8) are not present, implementations shall ignore writes to these bits and supply a zero value when read.
RVU.11.23	11.2 (p.65)	R	Standard software should preserve the contents of these bits (bits 31-8 of `fcsr`).
RVU.11.24	11.2 (p.65)	I	Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in *frm*.
RVU.11.25	11.2 (p.66) Table 11.1	R	Rounding mode encodings and meanings are listed below.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to nearest, ties to even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards −∞)
011	RUP	Round Up (towards +∞)
100	RMM	Round to Nearest, ties to Max Magnitude
101	-	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.

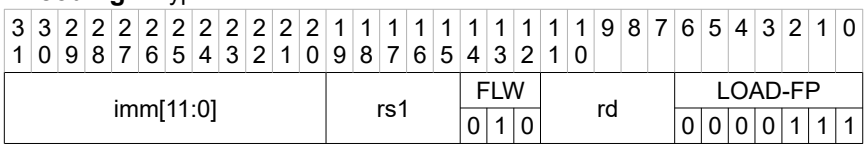
RVU.11.26	11.2 (p.65)	R	A value of 111 in the instruction’s *rm* field selects the dynamic rounding mode held in *frm*.
RVU.11.27	11.2 (p.65)	R	If frm is set to an invalid value (101–111), any subsequent attempt to execute a floating-point operation with a dynamic rounding mode will raise an illegal instruction exception.
RVU.11.28	11.2 (p.65)	R	Some instructions, including widening conversions, have the *rm* field but are nevertheless unaffected by the rounding mode; software should set their *rm* field to RNE (000).
RVU.11.29	11.2 (p.65)	C	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.
RVU.11.30	11.2 (p.65)	C	Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.
RVU.11.31	11.2 (p.66)	R	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

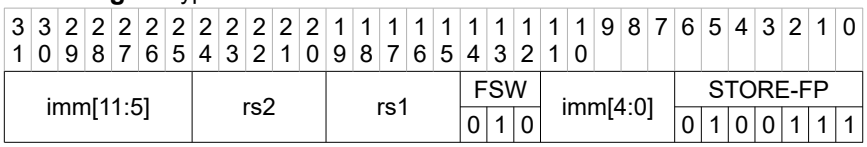
ID	REFERENCE	TYPE	DEFINITION
RVU.11.32	11.2 (p.66) Table 11.2	R	The accrued exception flags (*fflags*) meanings

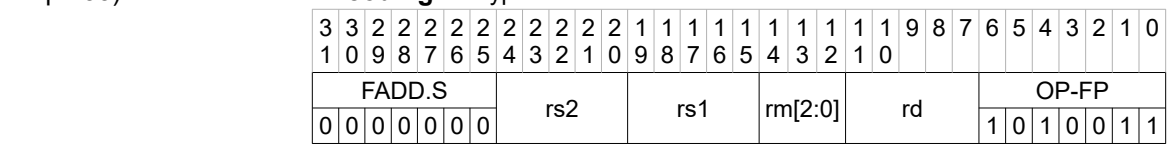
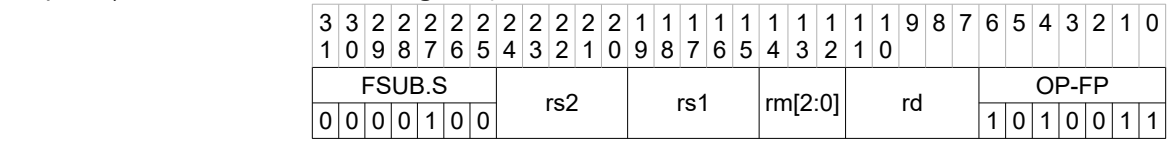
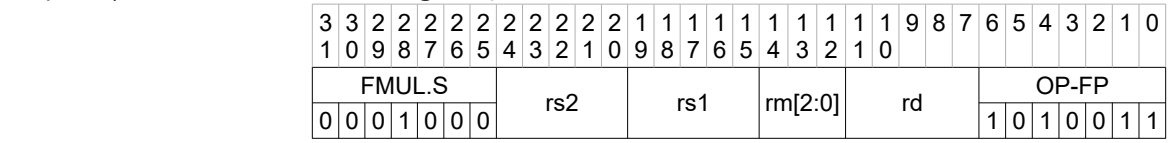
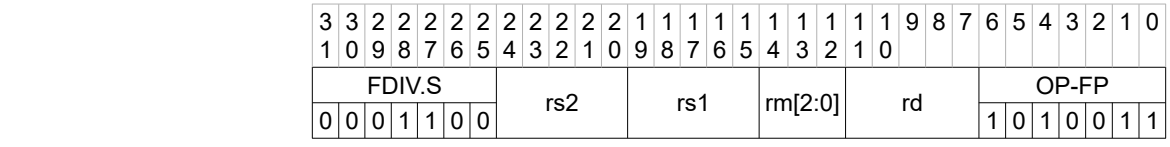
Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

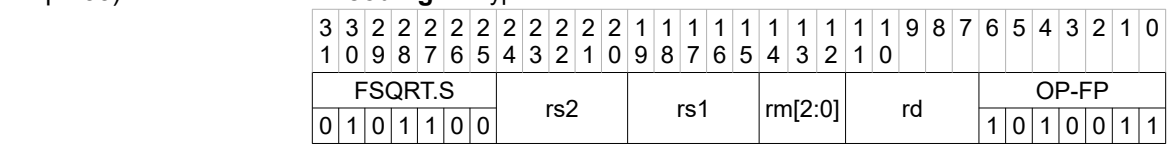
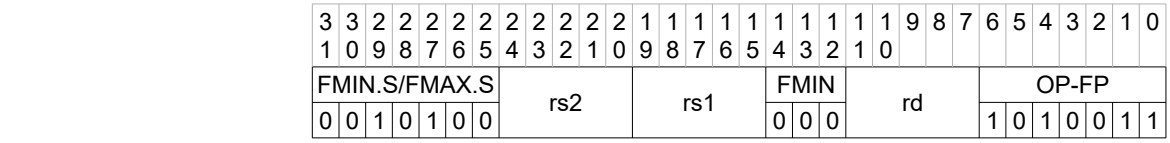
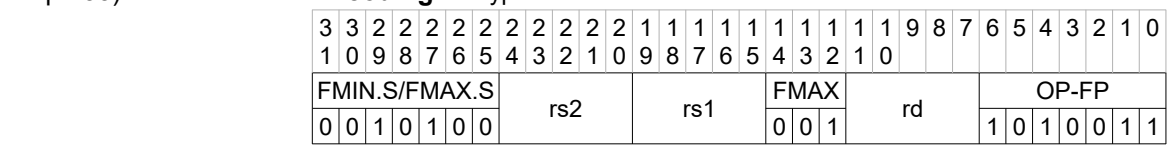
RVU.11.33	11.2 (p.66)	I	The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.
RVU.11.34	11.2 (p.66)	C	As allowed by the standard, we do not support traps on floating-point exceptions in the base ISA, but instead require explicit checks of the flags 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.
RVU.11.35	11.3 (p.66)	H	NaN Generation and Propagation
RVU.11.36	11.3 (p.66)	R	Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN.
RVU.11.37	11.3 (p.66)	I	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.
RVU.11.38	11.3 (p.66)	C	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.
RVU.11.39	11.3 (p.66)	C	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.
RVU.11.40	11.3 (p.66, p.67)	C	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.
RVU.11.41	11.4 (p.67)	H	Subnormal Arithmetic
RVU.11.42	11.4 (p.67)	R	Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard
RVU.11.43	11.4 (p.67)	R	In the parlance of the IEEE standard, tininess is detected after rounding
RVU.11.44	11.4 (p.67)	C	Detecting tininess after rounding results in fewer spurious underflow signals.
RVU.11.45	11.5 (p.67)	H	Single-Precision Load and Store Instructions
RVU.11.46	11.5 (p.67)	I	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.

ID	REFERENCE	TYPE	DEFINITION

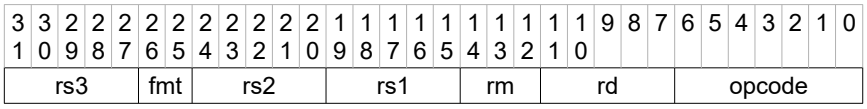
RVU.11.47	11.5 (p.67) & 24.0 (p.129, p.133)	R	FLW Instruction
			Load a single precision number from memory to register operation
			Encoding: I-Type
			 
			Valid Base: RV32, RV64
			Task: `mem_addr=x(rs1)+I-Immediate`
			`mem_val=MEM(mem_addr:mem_addr+3)`
			`f(rd)=mem_val;`
			Explanation: The memory address is obtained by adding integer register *rs1* to the I-Immediate value. It loads one 4 bytes (32-bit) single precision value from memory at the effective address and stores it into floating-point register *rd*.
			Special Case: none
			Exception: none

RVU.11.48	11.5 (p.67) & 24.0 (p.129, p.133)	R	FSW Instruction Store single precision number from register to memory instruction **Encoding**: S-Type  **Valid Base**: RV32 RV64 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr:mem_addr+3)=f(rs2)[31:0]` **Explanation**: The memory address is obtained by adding integer register *rs1* to the S-Immediate value. It stores the contents of floating-point register *rs2* into the memory at the effective address. **Special Case**: none **Exception**: none
RVU.11.49	11.5 (p.67)	R	FLW and FSW are only guaranteed to execute atomically if the effective address is naturally aligned.
RVU.11.50	11.5 (p.67)	R	FLWand FSW do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.
RVU.11.51	11.6 (p.67)	H	Single-Precision Floating-Point Computational Instructions
RVU.11.52	11.6 (p.67)	I	Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode.
RVU.11.53	11.6 (p.67)	R	All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in RVU.11.25.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.54	11.6 (p.68) & 24.0 (p.129, p.133)	R	FADD.S Instruction Single precision floating point addition operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)+f(rs2)` **Explanation**: FADD.S performs the single precision addition of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.11.55	11.6 (p.68) & 24.0 (p.129, p.133)	R	FSUB.S Instruction Single precision floating point subtraction operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)-f(rs2)` **Explanation**: FSUB.S performs the single precision subtraction of *rs2* from *rs1* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.11.56	11.6 (p.68) & 24.0 (p.129, p.133)	R	FMUL.S Instruction Single precision floating point multiplication operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)*f(rs2)` **Explanation**: FMUL.S performs the single precision multiplication of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.11.57	11.6 (p.68) & 24.0 (p.129, p.133)	R	FDIV.S Instructio Single precision floating point division operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)/f(rs2)` **Explanation**: FDIV.S performs the single precision division of *rs1* by *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none

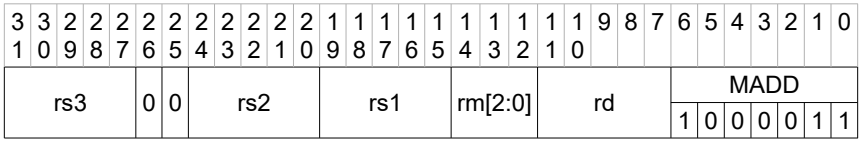
ID	REFERENCE	TYPE	DEFINITION
RVU.11.58	11.6 (p.68) & 24.0 (p.129, p.133)	R	FSQRT.S Instruction Single precision floating point square root operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=sqrt(f(rs1))` **Explanation**: FSQRT.S performs the single precision square root of *rs1* and writes the result into floating point register *rd*. *rs2* value is expected to be zero. **Special Case**: none **Exception**: none
RVU.11.59	11.6 (p.68) & 24.0 (p.129, p.133)	R	FMIN.S Instruction Single precision floating point minimum operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=min(f(rs1),f(rs2))` **Explanation**: FMIN.S performs the single precision minimum of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: -0.0 is considered smaller than +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. **Exception**: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.
RVU.11.60	11.6 (p.68) & 24.0 (p.129, p.133)	R	FMAX.S Instruction Single precision floating point maximum operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=max(f(rs1),f(rs2))` **Explanation**: FMAX.S performs the single precision maximum of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: -0.0 is considered smaller than +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. **Exception**: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.
RVU.11.61	11.6 (p.68)	C	Note that in version 2.2 of the F extension, the FMIN.S and FMAX.S instructions were amended to implement the proposed IEEE 754-201x minimumNumber and maximumNumber operations, rather than the IEEE 754-2008 minNum and maxNum operations. These operations differ in their handling of signaling NaNs.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.62	11.6 (p.68)	R	R4-Type instruction format for floating point fused muliply-add instructions



31–27	26–25	24–20	19–15	14–12	11–7	6–0
rs3	fmt	rs2	rs1	rm	rd	opcode

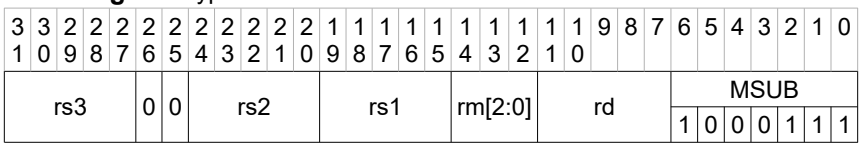
ID	REFERENCE	TYPE	DEFINITION
RVU.11.63	11.6 (p.68) & 24.0 (p.129, p.133)	R	FMADD.S Instruction Single precision multiply-add operation **Encoding**: R4-Type



31–27	26	25	24–20	19–15	14–12	11–7	6–0
rs3	0	0	rs2	rs1	rm[2:0]	rd	MADD 1000011

Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)*f(rs2)+f(rs3)`
Explanation: FMADD.S multiplies the values in *rs1* and *rs2*, adds the value in *rs3*, and writes the final result to *rd*.
Special Case: none
Exception: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

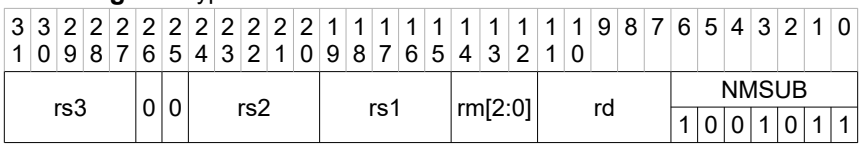
ID	REFERENCE	TYPE	DEFINITION
RVU.11.64	11.6 (p.68) & 24.0 (p.129, p.133)	R	FMSUB.S Instruction Single precision multiply-subtract operation **Encoding**: R4-Type



31–27	26	25	24–20	19–15	14–12	11–7	6–0
rs3	0	0	rs2	rs1	rm[2:0]	rd	MSUB 1000111

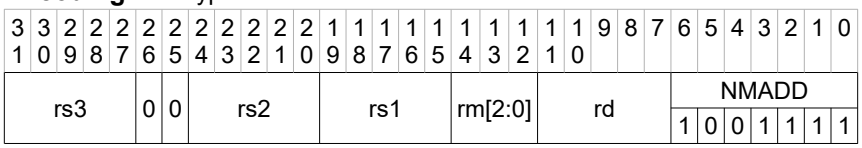
Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)*f(rs2)-f(rs3)`
Explanation: FMSUB.S multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*.
Special Case: none
Exception: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.65	11.6 (p.68) & 24.0 (p.129, p.133)	R	FNMSUB.S Instruction Single precision multiply-subtract negate operation **Encoding**: R4-Type



31–27	26	25	24–20	19–15	14–12	11–7	6–0
rs3	0	0	rs2	rs1	rm[2:0]	rd	NMSUB 1001011

Valid Base: RV32, RV64
Task: `f(rd)=-(f(rs1)*f(rs2))+f(rs3)`
Explanation: FNMSUB.S multiplies the values in *rs1* and *rs2*, negates the product, adds the value in *rs3*, and writes the final result to *rd*.
Special Case: none
Exception: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.66	11.6 (p.69) & 24.0 (p.129, p.133)	R	FNMADD.S Instruction Single precision multiply-add negate operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-(f(rs1)*f(rs2))-f(rs3)` **Explanation**: FNMADD.S multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.11.67	11.7 (p.69)	C	The FNMSUB and FNMADD instructions are counterintuitively named, owing to the naming of the corresponding instructions in MIPS-IV. The MIPS instructions were defined to negate the sum, rather than negating the product as the RISC-V instructions do, so the naming scheme was more rational at the time. The two definitions differ with respect to signed-zero results. The RISC-V definition matches the behavior of the x86 and ARM fused multiply-add instructions, but unfortunately the RISC-V FNMSUB and FNMADD instruction names are swapped compared to x86 and ARM.
RVU.11.68	11.7 (p.69)	C	The fused multiply-add (FMA) instructions consume a large part of the 32-bit instruction encoding space. Some alternatives considered were to restrict FMA to only use dynamic rounding modes, but static rounding modes are useful in code that exploits the lack of product rounding. Another alternative would have been to use rd to provide rs3, but this would require additional move instructions in some common sequences. The current design still leaves a large portion of the 32-bit encoding space open while avoiding having FMA be non-orthogonal.
RVU.11.69	11.7 (p.69)	C	The IEEE 754-2008 standard permits, but does not require, raising the invalid exception for the operation ∞× 0 + qNaN.
RVU.11.70	11.7 (p.69)	H	Single-Precision Floating-Point Conversion and Move Instructions
RVU.11.71	11.7 (p.69)	I	Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.72	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.W.S Instruction Single precision floating point to 32-bit integer conversion operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.W.S												rs1					rm[2:0]			rd			OP-FP							
1	1	0	0	0	0	0	0	0	0	0	0														1	0	1	0	0	1	1

Valid Base: RV32, RV64
Task: `x(rd)=F2INT(f(rs1))`
Explanation: FCVT.W.S converts a single precision floating-point number in floating point register *rs1* to a signed 32-bit integer in integer register *rd*. If XLEN>32, result is sign extended. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < -2^{31}, x(*rd*) = -2^{31}
If f(*rs1*) > $2^{31}-1$ or NaN, x(*rd*) = $2^{31}-1$
Exception: If f(*rs1*) < -2^{31}, invalid flag is set
If f(*rs1*) > $2^{31}-1$ or NaN, invalid flag is set

ID	REFERENCE	TYPE	DEFINITION
RVU.11.73	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.L.S Instruction Single precision floating point to 64-bit integer conversion operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.L.S												rs1					rm[2:0]			rd			OP-FP							
1	1	0	0	0	0	0	0	0	0	0	1														1	0	1	0	0	1	1

Valid Base: RV64
Task: `x(rd)=F2INT(f(rs1))`
Explanation: FCVT.L.S converts a single precision floating-point number in floating point register *rs1* to a signed 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < -2^{63}, x(*rd*) = -2^{63}
If f(*rs1*) > $2^{63}-1$ or NaN, x(*rd*) = $2^{63}-1$
Exception: If f(*rs1*) < -2^{63}, invalid flag is set
If f(*rs1*) > $2^{63}-1$ or NaN, invalid flag is set

ID	REFERENCE	TYPE	DEFINITION
RVU.11.74	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.WU.S Instruction Single precision floating point to unsigned 32-bit integer conversion **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.WU.S												rs1					rm[2:0]			rd			OP-FP							
1	1	0	0	0	0	0	0	0	0	0	1														1	0	1	0	0	1	1

Valid Base: RV32, RV64
Task: `x(rd)=F2UINT(f(rs1))`
Explanation: FCVT.WU.S converts a single precision floating-point number in floating point register *rs1* to a unsigned 32-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < 0, x(*rd*) = 0
If f(*rs1*) > $2^{31}-1$ or NaN, x(*rd*) = $2^{31}-1$
Exception: If f(*rs1*) < 0, invalid flag is set
If f(*rs1*) > $2^{31}-1$ or NaN, invalid flag is set

ID	REFERENCE	TYPE	DEFINITION
RVU.11.75	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.LU.S Instruction Single precision floating point to unsigned 64-bit integer conversion **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.LU.S												rs1					rm[2:0]			rd					OP-FP						
1	1	0	0	0	0	0	0	0	0	1	1														1	0	1	0	0	1	1

Valid Base: RV64
Task: `x(rd)=F2UINT(f(rs1))`
Explanation: FCVT.LU.S converts a single precision floating-point number in floating point register *rs1* to an unsigned 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < 0, x(*rd*) = 0
If f(*rs1*) > 2^{63}-1 or NaN, x(*rd*) = 2^{63}-1
Exception: If f(*rs1*) < 0, invalid flag is set
If f(*rs1*) > 2^{63}-1 or NaN, invalid flag is set

RVU.11.76	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.S.W Instruction 32-bit integer to single precision floating point conversion operation **Encoding**: R-Type

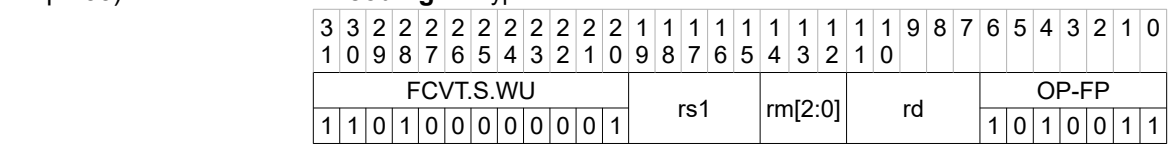
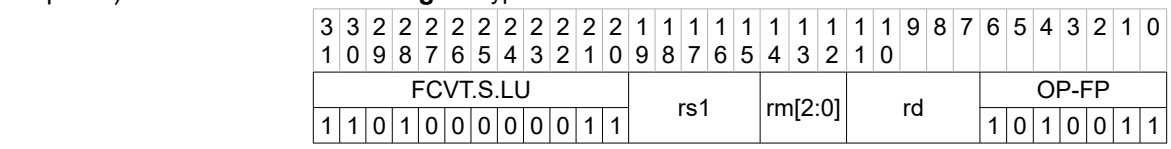
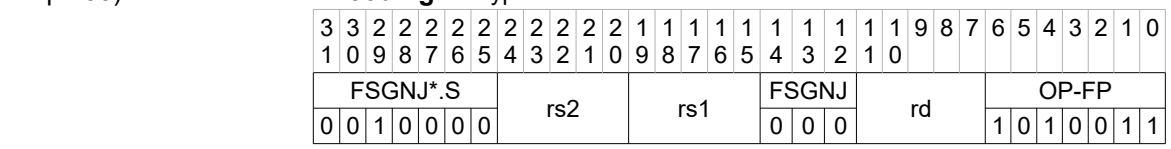
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.S.W												rs1					rm[2:0]			rd					OP-FP						
1	1	0	1	0	0	0	0	0	0	0	0														1	0	1	0	0	1	1

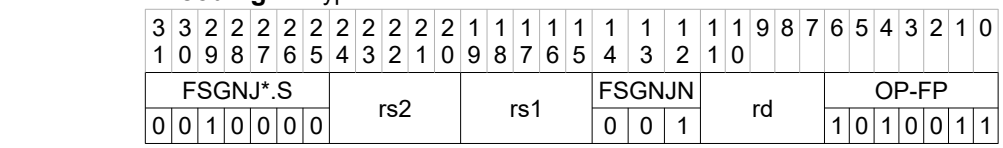
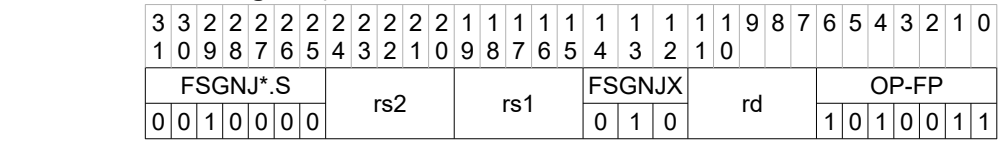
Valid Base: RV32, RV64
Task: `f(rd)=INT2F(x(rs1))`
Explanation: FCVT.S.W converts a 32 bit signed integer number in integer register *rs1* to a floating-point number in floating point register *rd*. f(*rd*) is rounded according to rm field.
Special Case: none
Exception: none

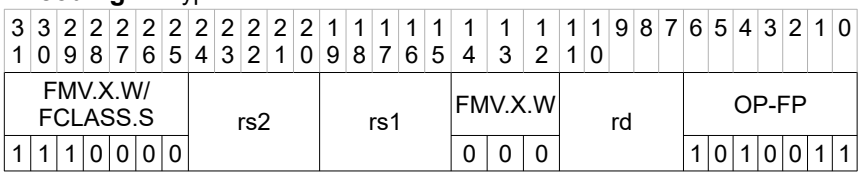
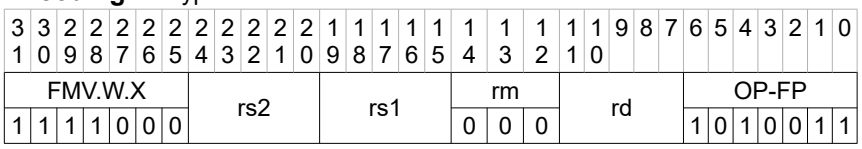
RVU.11.77	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.S.L Instruction 64-bit integer to single precision floating point conversion operation **Encoding**: R-Type

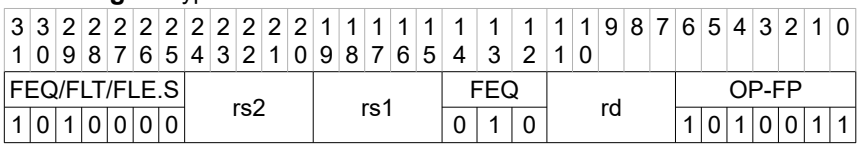
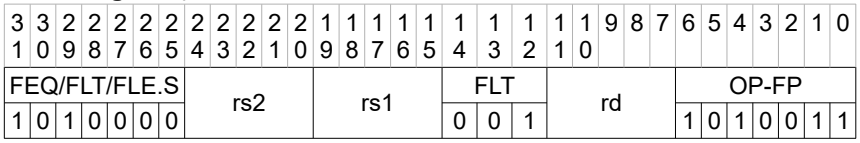
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.S.L												rs1					rm[2:0]			rd					OP-FP						
1	1	0	1	0	0	0	0	0	0	1	0														1	0	1	0	0	1	1

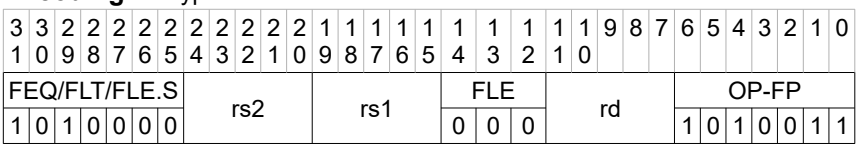
Valid Base: RV64
Task: `f(rd)=INT2F(x(rs1))`
Explanation: FCVT.S.L converts a 64-bit signed integer number in integer register *rs1* to a floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field.
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.11.78	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.S.WU Instruction 32-bit unsigned integer to single precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=UINT2F(x(rs1))` **Explanation**: FCVT.S.WU converts a 32 bit unsigned integer number in integer register *rs1* to a floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.11.79	11.7 (p.69) & 24.0 (p.129, p.133)	R	FCVT.S.LU Instruction 64-bit unsigned integer to single precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `f(rd)=UINT2F(x(rs1))` **Explanation**: FCVT.S.LU converts a 64-bit unsigned integer number in integer register *rs1* to a floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.11.80	11.7 (p.70)	R	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*.
RVU.11.81	11.7 (p.70)	R	For FSGNJ, the result's sign bit is *rs2*'s sign bit;
RVU.11.82	11.7 (p.70)	R	… for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit;
RVU.11.83	11.7 (p.70)	R	… for FSGNJX, the sign bit is the XOR of the sign bits of *rs1* and *rs2*.
RVU.11.84	11.7 (p.70) & 24.0 (p.129, p.133)	R	FSGNJ.S Instruction Change sign to register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJ.S changes the sign of the number in floating point register at *rs1* to the sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to sign bit of *rs2*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.11.85	11.7 (p.70) & 24.0 (p.129, p.133)	R	FSGNJN.S Instruction Change sign to opposite of register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-1*SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJN.S changes the sign of the number in floating point register at *rs1* to the opposite sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the not of the sign bit of *rs2*. **Special Case**: none **Exception**: none
RVU.11.86	11.7 (p.70) & 24.0 (p.129, p.133)	R	FSGNJX.S Instruction Change sign to multiplication of signs operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=SGN(f(rs2))*SGN(f(rs1))*|f(rs1)|` **Explanation**: FSGNJX.S changes the sign of the number in floating point register at *rs1* to the multiplication of the sign of the number in floating point register at *rs1* and *rs2*, and then stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the xor of sign bit of *rs1* and sign bit of *rs2*. **Special Case**: none **Exception**: none
RVU.11.87	11.7 (p.70)	R	Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.
RVU.11.88	11.7 (p.70)	I	Note, `FSGNJ.S rx, ry, ry` moves *ry* to *rx* ...; `FSGNJN.S rx, ry, ry` moves the negation of *ry* to *rx* ...; and `FSGNJX.S rx, ry, ry` moves the absolute value of *ry* to *rx* ...
RVU.11.89	11.7 (p.70)	C	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.90	11.7 (p.70) & 24.0 (p.129, p.133)	R	FMV.X.W Instruction Move single precision floating point register to integer register operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)[31:0]=f(rs1);` `x(rd)[XLEN-1:32]=signbit(f(rs1))` **Explanation**: FMV.X.W moves the single-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the lower 32 bits of integer register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit. *rs2* is not used and expected to be zero. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.11.91	11.7 (p.70) & 24.0 (p.129, p.133)	R	FMV.W.X Instruction Move integer register to single precision floating point register operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=x(rs1)[31:0];` **Explanation**: 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. *rs2* is not used and expected to be zero. **Special Case**: none **Exception**: none
RVU.11.92	11.7 (p.71)	C	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.
RVU.11.93	11.7 (p.71)	C	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.
RVU.11.94	11.8 (p.71)	H	Single-Precision Floating-Point Compare Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.11.95	11.8 (p.71)	I	Floating-point compare instructions (FEQ.S, FLT.S, FLE.S) perform the specified comparison between floating-point registers (*rs1* = *rs2*, *rs1* < *rs2*, *rs1* ≤ *rs2*) writing 1 to the integer register *rd* if the condition holds, and 0 otherwise.
RVU.11.96	11.8 (p.71) & 24.0 (p.129, p.133)	R	FEQ.S Instruction Single precision floating point equality check operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)=f(rs2); x(rd)=0 otherwise` **Explanation**: FEQ.S performs the equality comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FEQ.S performs a quiet comparison: it only sets the invalid operation exception flag if either input is a signaling NaN.
RVU.11.97	11.8 (p.71) & 24.0 (p.129, p.133)	R	FLT.S Instruction Single precision floating point less than comparison operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)<f(rs2); x(rd)=0 otherwise` **Explanation**: FLT.S performs the less than comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FLT.S performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.

ID	REFERENCE	TYPE	DEFINITION
RVU.11.98	11.8 (p.71) & 24.0 (p.129, p.133)	R	FLE.S Instruction Single precision floating point less than comparison operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)≤f(rs2); x(rd)=0 otherwise` **Explanation**: FLE.S performs the less than and equal to comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FLE.S performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.
RVU.11.99	11.8 (p.71)	C	The F extension provides a ≤ comparison, whereas the base ISA provides a ≥ branch comparison. Because ≤ can be synthesized from ≥ and vice-versa, there is no performance implication to this inconsistency, but it is nevertheless an unfortunate incongruity in the ISA.
RVU.11.100	11.9 (p.71)	H	Single-Precision Floating-Point Classify Instruction

ID	REFERENCE	TYPE	DEFINITION
RVU.11.101	11.9 (p.71, p.72) & 24.0 (p.129, p.133) Table 11.5	R	FCLASS.S Instruction Single precision floating point classify operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMV.X.W/ FCLASS.S							rs2					rs1					FCLASS			rd								OP-FP			
1	1	1	0	0	0	0											0	0	1									1	0	1	0	0	1	1

Valid Base: RV32, RV64

Task: `x(rd)= CLASS(f(rs1))`

Explanation: 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. *rs2* is not used and expected to be zero. The format of the mask is described in Table below. 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.

***rd* bit**	**Meaning**
0	Rs1 is −∞
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 +∞.
8	rs1 is a signaling NaN.
9	rs1 is a quiet NaN.

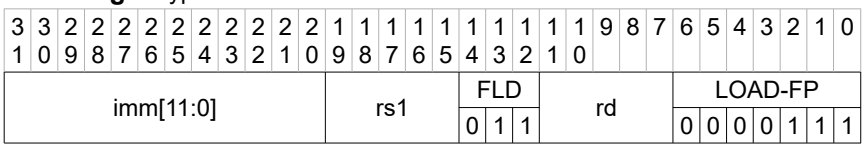
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Exception: none

CHAPTER 12 “D” Standard Extension for Double-Precision Floating-Point

ID	REFERENCE	TYPE	DEFINITION
RVU.12.1	12.0 (p.73)	H	“D” Standard Extension for Double-Precision Floating-Point
RVU.12.2	12.0 (p.73) preface (p.i)	I	“D” extension version is 2.2 and status is ratified.
RVU.12.3	12.0 (p.73)	I	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.
RVU.12.4	12.0 (p.73)	R	The D extension depends on the base single-precision instruction subset F.
RVU.12.5	12.1 (p.73)	R	D Register State
RVU.12.6	12.1 (p.73)	R	The D extension widens the 32 floating-point registers, f0–f31, to 64 bits (FLEN=64).
RVU.12.7	12.1 (p.73)	R	The f registers can now hold either 32-bit or 64-bit floating-point values.
RVU.12.8	12.1 (p.73)	C	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.
RVU.12.9	12.2 (p.73)	H	NaN Boxing of Narrower Values
RVU.12.10	12.2 (p.73)	R	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.
RVU.12.11	12.2 (p.73)	R	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 ≤ FLEN.
RVU.12.12	12.2 (p.73)	R	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.
RVU.12.13	12.2 (p.74)	C	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.
RVU.12.14	12.2 (p.74)	R	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).
RVU.12.15	12.2 (p.74)	R	A narrower n-bit transfer, n < FLEN, into the f registers will create a valid NaN-boxed value.
RVU.12.16	12.2 (p.74)	R	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.
RVU.12.17	12.2 (p.74)	R	Apart from transfer operations (FLn/FSn and FMV.n.X/FMV.X.n), 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.

ID	REFERENCE	TYPE	DEFINITION
RVU.12.18	12.2 (p.74)	C	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.
RVU.12.19	12.2 (p.74)	C	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.
RVU.12.20	12.2 (p.74)	C	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.
RVU.12.21	12.3 (p.74)	H	Double-Precision Load and Store Instructions
RVU.12.22	12.3 (p.74, p.75) & 24.0 (p.129, p.134)	R	FLD Instruction Load a double precision number from memory to register operation **Encoding**: I-Type



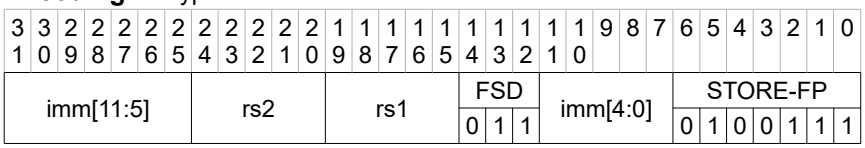
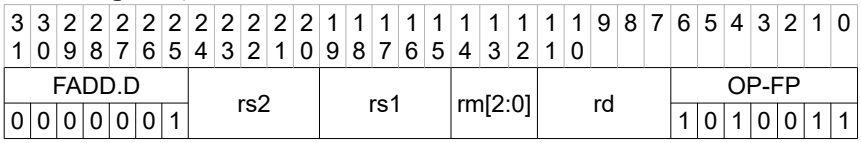
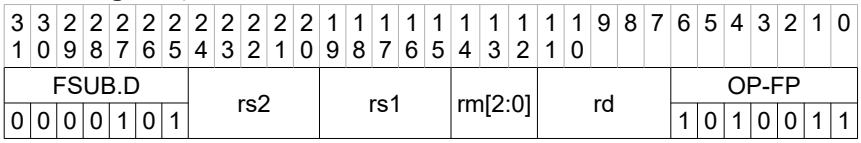
Valid Base: RV32, RV64

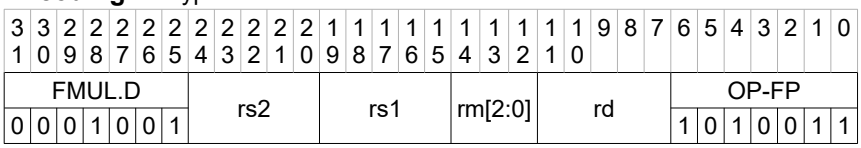
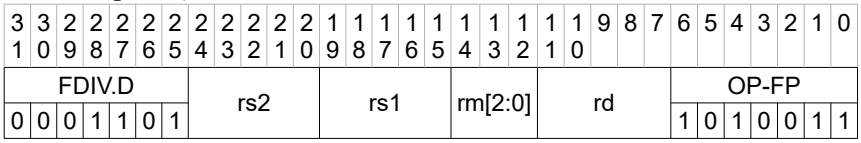
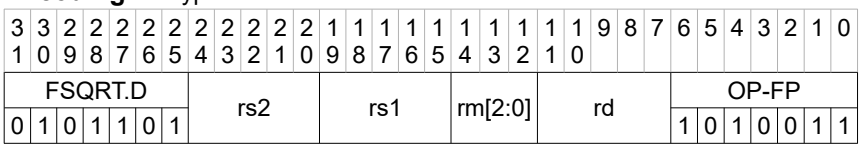
Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr:mem_addr+7)
f(rd)=mem_val;
```

Explanation: The memory address is obtained by adding integer register *rs1* to the I-Immediate value. It loads one (64-bit) double precision value from memory at the effective address and stores it into floating-point register *rd*.

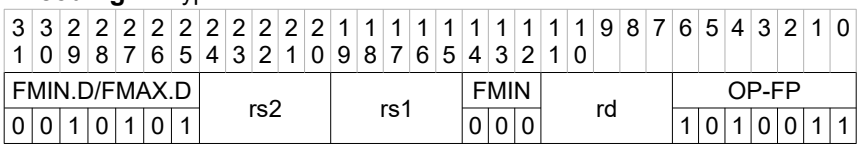
Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.12.23	12.3 (p.75) & 24.0 (p.129, p.134)	R	FSD Instruction Store double precision number from register to memory operation **Encoding**: S-Type  **Valid Base**: RV32 RV64 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr:mem_addr+7)=f(rs2)[63:0]` **Explanation**: The memory address is obtained by adding integer register *rs1* to the S-Immediate value. It stores the contents of floating-point register *rs2* (64 bits) into the memory at the effective address. **Special Case**: none **Exception**: none
RVU.12.24	12.3 (p.75)	R	FLD and FSD are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64.
RVU.12.25	12.3 (p.75)	R	FLD and FSD do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.
RVU.12.26	12.4 (p.75)	H	Double-Precision Floating-Point Computational Instructions
RVU.12.27	12.4 (p.75)	R	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.
RVU.12.28	12.4 (p.75) & 24.0 (p.129, p.134)	R	FADD.D Instruction Double precision floating point addition operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)+f(rs2)` **Explanation**: FADD.D performs the double precision addition of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.12.29	12.4 (p.75) & 24.0 (p.129, p.134)	R	FSUB.D Instruction Double precision floating point subtraction operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)-f(rs2)` **Explanation**: FSUB.D performs the double precision subtraction of *rs2* from *rs1* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.12.30	12.4 (p.75) & 24.0 (p.129, p.134)	R	FMUL.D Instruction Double precision floating point multiplication operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)*f(rs2)` **Explanation**: FMUL.D performs the double precision multiplication of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.12.31	12.4 (p.75) & 24.0 (p.129, p.134)	R	FDIV.D Instruction Double precision floating point division operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)/f(rs2)` **Explanation**: FDIV.D performs the double precision division of *rs1* by *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.12.32	12.4 (p.75) & 24.0 (p.129, p.134)	R	FSQRT.D Instruction Double precision floating point square root operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=sqrt(f(rs1))` **Explanation**: FSQRT.D performs the double precision square root of *rs1* and writes the result into floating point register *rd*. *rs2* value is expected to be zero. **Special Case**: none **Exception**: none

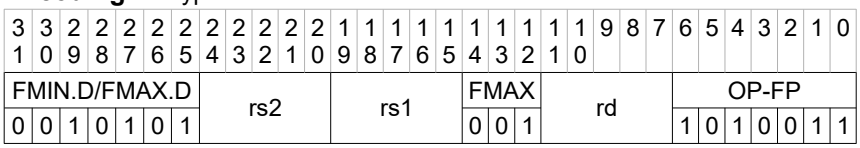
ID	REFERENCE	TYPE	DEFINITION
RVU.12.33	12.4 (p.75) & 24.0 (p.129, p.134)	R	FMIN.D Instruction Double precision floating point minimum operation **Encoding**: R-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMIN.D/FMAX.D							rs2					rs1					FMIN			rd						OP-FP						
0	0	1	0	1	0	1											0	0	0							1	0	1	0	0	1	1

Valid Base: RV32, RV64
Task: `f(rd)=min(f(rs1),f(rs2))`
Explanation: FMIN.D performs the double precision minimum of *rs1* and *rs2* and writes the result into floating point register *rd*.
Special Case: -0.0 is considered smaller than +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.
Exception: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

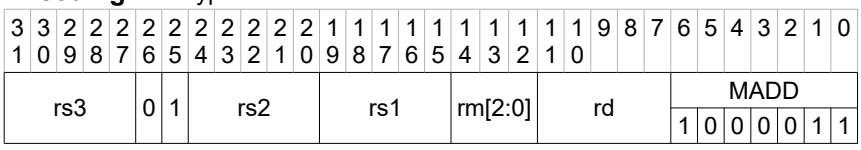
RVU.12.34	12.4 (p.75) & 24.0 (p.129, p.134)	R	FMAX.D Instruction Double precision floating point maximum operation **Encoding**: R-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMIN.D/FMAX.D							rs2					rs1					FMAX			rd						OP-FP						
0	0	1	0	1	0	1											0	0	1							1	0	1	0	0	1	1

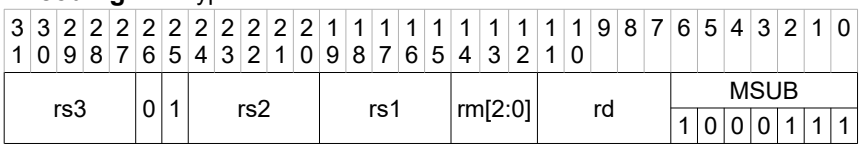
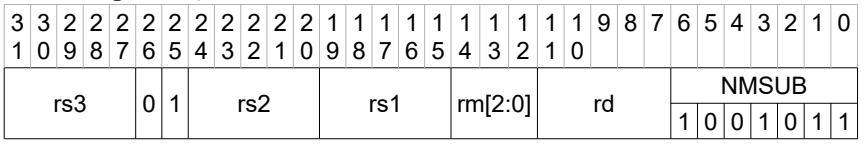
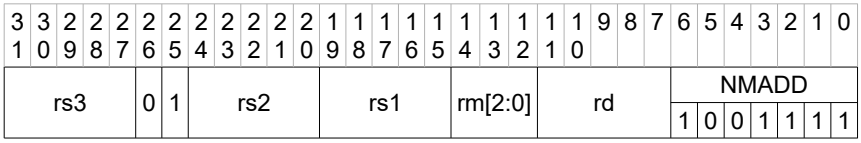
Valid Base: RV32, RV64
Task: `f(rd)=max(f(rs1),f(rs2))`
Explanation: FMAX.D performs the double precision maximum of *rs1* and *rs2* and writes the result into floating point register *rd*.
Special Case: -0.0 is considered smaller than +0.0
If both inputs are NaNs, the result is the canonical NaN.
If only one operand is a NaN, the result is the non-NaN operand.
Exception: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

RVU.12.35	12.4 (p.75) & 24.0 (p.129, p.134)	R	FMADD.D Instruction Double precision multiply-add operation **Encoding**: R4-Type



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rs3					0	1	rs2					rs1					rm[2:0]			rd						MADD						
																									1	0	0	0	0	1	1

Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)*f(rs2)+f(rs3)`
Explanation: FMADD.D multiplies the values in *rs1* and *rs2*, adds the value in *rs3*, and writes the final result to *rd*.
Special Case: none
Exception: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

ID	REFERENCE	TYPE	DEFINITION
RVU.12.36	12.4 (p.75) & 24.0 (p.129, p.134)	R	FMSUB.D Instruction Double precision multiply-subtract operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)*f(rs2)-f(rs3)` **Explanation**: FMSUB.D multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.12.37	12.4 (p.75) & 24.0 (p.129, p.134)	R	FNMSUB.D Instruction Double precision multiply-subtract negate operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-(f(rs1)*f(rs2))+f(rs3)` **Explanation**: FNMSUB.D multiplies the values in *rs1* and *rs2*, negates the product, adds the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.12.38	12.4 (p.75) & 24.0 (p.129, p.134)	R	FNMADD.D Instruction Double precision multiply-add negate operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-(f(rs1)*f(rs2))-f(rs3)` **Explanation**: FNMADD.D multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.12.39	12.5 (p.75)	H	Double-Precision Floating-Point Conversion and Move Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.12.40	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.W.D Instruction Double precision floating point to 32-bit integer conversion operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.W.D												rs1					rm[2:0]			rd					OP-FP						
1	1	0	0	0	0	1	0	0	0	0	0														1	0	1	0	0	1	1

Valid Base: RV32, RV64
Task: `x(rd)=D2INT(f(rs1))`
Explanation: FCVT.W.D converts a double precision floating-point number in floating point register *rs1* to a signed 32-bit integer in integer register *rd*. If XLEN>32, result is sign extended. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < -2^{31}, x(*rd*) = -2^{31}
If f(*rs1*) > $2^{31}-1$ or NaN, x(*rd*) = $2^{31}-1$
Exception: If f(*rs1*) < -2^{31}, invalid flag is set
If f(*rs1*) > $2^{31}-1$ or NaN, invalid flag is set

ID	REFERENCE	TYPE	DEFINITION
RVU.12.41	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.L.D Instruction Double precision floating point to 64-bit integer conversion operation **Encoding**: R-Type

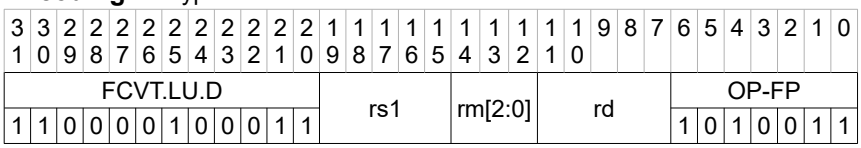
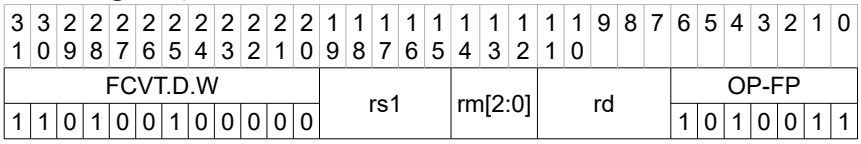
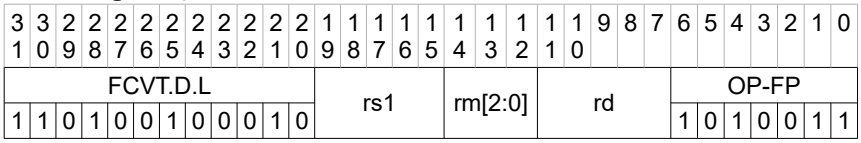
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.L.D												rs1					rm[2:0]			rd					OP-FP						
1	1	0	0	0	0	1	0	0	0	1	0														1	0	1	0	0	1	1

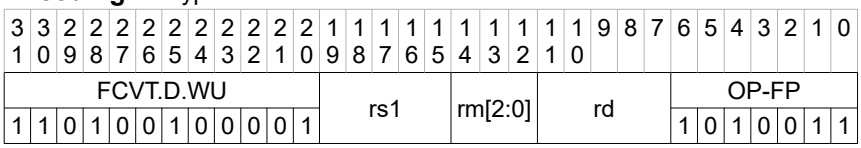
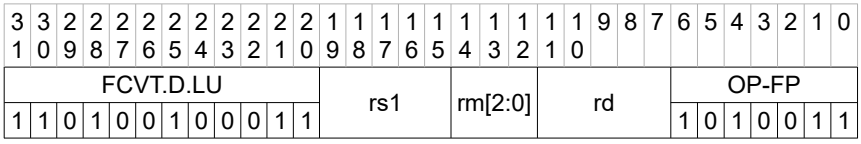
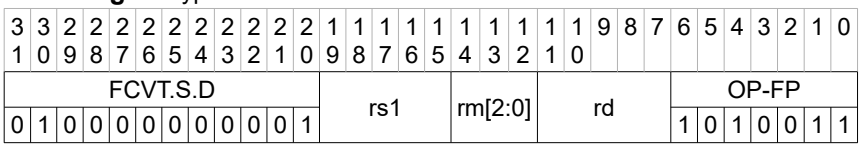
Valid Base: RV64
Task: `x(rd)=D2INT(f(rs1))`
Explanation: FCVT.L.D converts a double precision floating-point number in floating point register *rs1* to a signed 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < -2^{63}, x(*rd*) = -2^{63}
If f(*rs1*) > $2^{63}-1$ or NaN, x(*rd*) = $2^{63}-1$
Exception: If f(*rs1*) < -2^{63}, invalid flag is set
If f(*rs1*) > $2^{63}-1$ or NaN, invalid flag is set

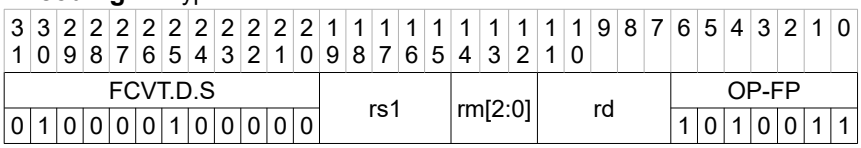
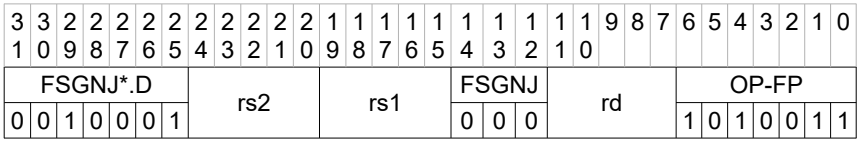
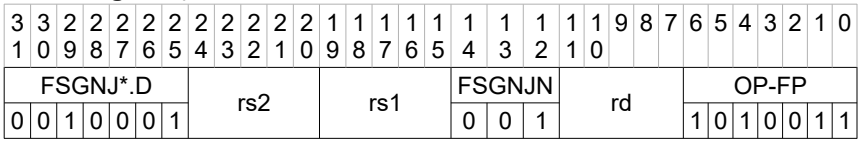
ID	REFERENCE	TYPE	DEFINITION
RVU.12.42	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.WU.D Instruction Double precision floating point to unsigned 32-bit integer conversion **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FCVT.WU.D												rs1					rm[2:0]			rd					OP-FP						
1	1	0	0	0	0	1	0	0	0	0	1														1	0	1	0	0	1	1

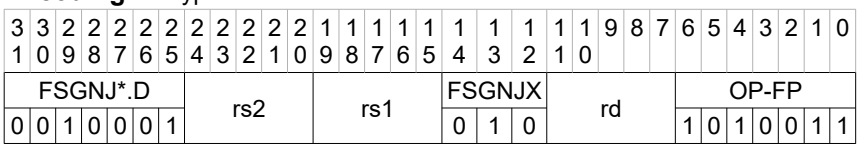
Valid Base: RV32, RV64
Task: `x(rd)=D2UINT(f(rs1))`
Explanation: FCVT.WU.D converts a double precision floating-point number in floating point register *rs1* to an unsigned 32-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
If f(*rs1*) < 0, x(*rd*) = 0
If f(*rs1*) > $2^{31}-1$ or NaN, x(*rd*) = $2^{31}-1$
Exception: If f(*rs1*) < 0, invalid flag is set
If f(*rs1*) > $2^{31}-1$ or NaN, invalid flag is set

ID	REFERENCE	TYPE	DEFINITION
RVU.12.43	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.LU.D Instruction Double precision floating point to unsigned 64-bit integer conversion **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `x(rd)=D2UINT(f(rs1))` **Explanation**: FCVT.LU.D converts a double precision floating-point number in floating point register *rs1* to an unsigned 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. If f(*rs1*) < 0, x(*rd*) = 0 If f(*rs1*) > 2^{63}-1 or NaN, x(*rd*) = 2^{63}-1 **Exception**: If f(*rs1*) < 0, invalid flag is set If f(*rs1*) > 2^{63}-1 or NaN, invalid flag is set
RVU.12.44	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.D.W Instruction 32-bit integer to double precision floating point conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=INT2D(x(rs1))` **Explanation**: FCVT.D.W converts a 32 bit signed integer number in integer register *rs1* to a double precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to rm field. **Special Case**: none **Exception**: none
RVU.12.45	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.D.L Istruction 64-bit integer to double precision floating point conversion operation **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `f(rd)=INT2D(x(rs1))` **Explanation**: FCVT.D.L converts a 64-bit signed integer number in integer register *rs1* to a double precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.12.46	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.D.WU Instruction 32-bit unsigned integer to double precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=UINT2D(x(rs1))` **Explanation**: FCVT.D.WU converts a 32 bit unsigned integer number in integer register *rs1* to a double-precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.12.47	12.5 (p.75, p.76) & 24.0 (p.129, p.134)	R	FCVT.D.LU Instruction 64-bit unsigned integer to double precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `f(rd)=UINT2D(x(rs1))` **Explanation**: FCVT.D.LU converts a 64-bit unsigned integer number in integer register *rs1* to a double-precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.12.48	12.5 (p.76) & 24.0 (p.129, p.134)	R	FCVT.S.D Instruction Double precision floating point to single precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=D2F(f(rs1))` **Explanation**: FCVT.S.D converts a double precision floating point number in *rs1* to a single precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none

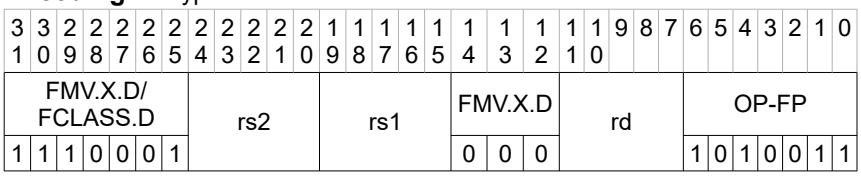
ID	REFERENCE	TYPE	DEFINITION
RVU.12.49	12.5 (p.76) & 24.0 (p.129, p.134)	R	FCVT.D.S Instruction Single precision floating point to double precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=F2D(f(rs1))` **Explanation**: FCVT.D.S converts a single precision floating point number in *rs1* to a double precision floating-point number in floating point register *rd*. **Special Case**: none **Exception**: none
RVU.12.50	12.5 (p.76) & 24.0 (p.129, p.134)	R	FSGNJ.D Instruction Change sign to register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJ.D changes the sign of the number in floating point register at *rs1* to the sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to sign bit of *rs2*. **Special Case**: none **Exception**: none
RVU.12.51	12.5 (p.76) & 24.0 (p.129, p.134)	R	FSGNJN.D Instruction Change sign to opposite of register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-1*SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJN.D changes the sign of the number in floating point register at *rs1* to the opposite sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the not of the sign bit of *rs2*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.12.52	12.5 (p.76) & 24.0 (p.129, p.134)	R	FSGNJX.D Instruction Change sign to multiplication of signs operation **Encoding**: R-Type



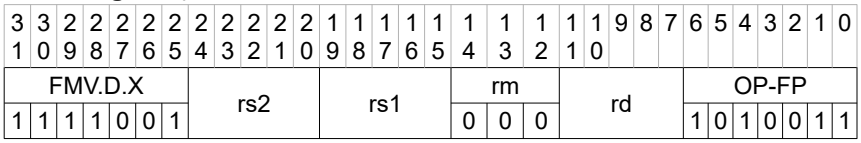
Valid Base: RV32, RV64
Task: `f(rd)=SGN(f(rs2))*SGN(f(rs1))*|f(rs1)|`
Explanation: FSGNJX.D changes the sign of the number in floating point register at *rs1* to the multiplication of the sign of the number in floating point register at *rs1* and *rs2*, and then stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the xor of sign bit of *rs1* and sign bit of *rs2*.
Special Case: none
Exception: none

RVU.12.53	12.5 (p.76) & 24.0 (p.129, p.134)	R	FMV.X.D Instruction Move double precision floating point register to integer register operation **Encoding**: R-Type

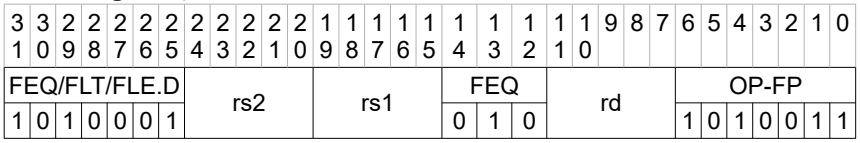
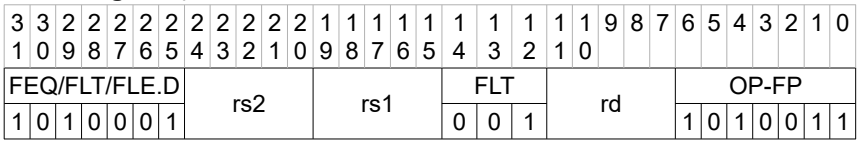


Valid Base: RV64
Task: `x(rd)[63:0]=f(rs1);`
Explanation: FMV.X.D moves the double-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the 64 bit integer register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. *rs2* is not used and expected to be zero.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

RVU.12.54	12.5 (p.76) & 24.0 (p.129, p.134)	R	FMV.D.X Instruction Move integer register to double precision floating point register operation **Encoding**: R-Type



Valid Base: RV64
Task: `f(rd)=x(rs1)[63:0];`
Explanation: FMV.D.X moves the double-precision value encoded in IEEE 754-2008 standard encoding from 64 bits 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. *rs2* is not used and expected to be zero.
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.12.55	12.5 (p.77)	C	Early versions of the RISC-V ISA had additional instructions to allow RV32 systems to transfer between the upper and lower portions of a 64-bit floating-point register and an integer register. However, these would be the only instructions with partial register writes and would add complexity in implementations with recoded floating-point or register renaming, requiring a pipeline read-modify-write sequence. Scaling up to handling quad-precision for RV32 and RV64 would also require additional instructions if they were to follow this pattern. The ISA was defined to reduce the number of explicit int-float register moves, by having conversions and comparisons write results to the appropriate register file, so we expect the benefit of these instructions to be lower than for other ISAs. We note that for systems that implement a 64-bit floating-point unit including fused multiply-add support and 64-bit floating-point loads and stores, the marginal hardware cost of moving from a 32-bit to a 64-bit integer datapath is low, and a software ABI supporting 32-bit wide address-space and pointers can be used to avoid growth of static data and dynamic memory traffic.
RVU.12.56	12.6 (p.77)	H	Double-Precision Floating-Point Compare Instructions
RVU.12.57	12.6 (p.77) & 24.0 (p.129, p.134)	R	FEQ.D Instruction Double precision floating point equality check operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)=f(rs2); x(rd)=0 otherwise` **Explanation**: FEQ.D performs the equality comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FEQ.D performs a quiet comparison: it only sets the invalid operation exception flag if either input is a signaling NaN.
RVU.12.58	12.6 (p.77) & 24.0 (p.129, p.134)	R	FLT.D Instruction Double precision floating point less than comparison operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)<f(rs2); x(rd)=0 otherwise` **Explanation**: FLT.D performs the less than comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FLT.D performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.

ID	REFERENCE	TYPE	DEFINITION
RVU.12.59	12.6 (p.77) & 24.0 (p.129, p.134)	R	FLE.D: Double precision floating point less than comparison operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FEQ/FLT/FLE.D							rs2					rs1					FLE			rd			OP-FP								
1	0	1	0	0	0	1											0	0	0				1	0	1	0	0	1	1		

Valid Base: RV32, RV64
Task: `x(rd)=1 when f(rs1)≤f(rs2); x(rd)=0 otherwise`
Explanation: FLE.D performs the less than and equal to comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise.
Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
The result is 0 if either operand is NaN.
Exception: FLE.D performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.

RVU.12.60	12.7 (p.77)	H	Double-Precision Floating-Point Classify Instruction
RVU.12.61	12.7 (p.77) & 24.0 (p.129, p.134) Table 11.5	R	FCLASS.D: Double precision floating point classify operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMV.X.D/ FCLASS.D							rs2					rs1					FCLASS			rd			OP-FP								
1	1	1	0	0	0	1											0	0	1				1	0	1	0	0	1	1		

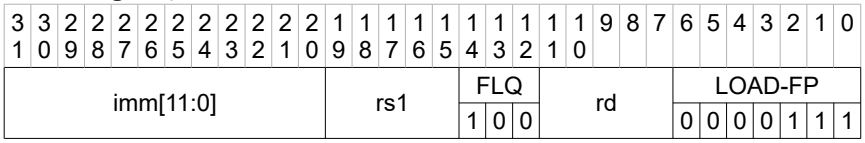
Valid Base: RV32, RV64
Task: `x(rd)= CLASS(f(rs1))`
Explanation: The FCLASS.D instruction examines the value in floating-point register *rs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. *rs2* is not used and expected to be zero. The format of the mask is described in Table below. 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.

***rd* bit**	**Meaning**
0	Rs1 is −∞
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 +∞.
8	rs1 is a signaling NaN.
9	rs1 is a quiet NaN.

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.
Exception: none

CHAPTER 13 “Q” Standard Extension for Quad-Precision Floating-Point

ID	REFERENCE	TYPE	DEFINITION
RVU.13.1	13.0 (p.79)	H	“Q” Standard Extension for Quad-Precision Floating-Point
RVU.13.2	13.0 (p.79) preface (p.i)	I	“Q” extension version is 2.2 and status is ratified.
RVU.13.3	13.0 (p.79)	I	This chapter describes the Q standard extension for 128-bit quad-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard.
RVU.13.4	13.0 (p.79)	R	The quad-precision binary floating-point instruction-set extension is named “Q”;...
RVU.13.5	13.0 (p.79)	R	.. it (Q extension) depends on the double-precision floating point extension D
RVU.13.6	13.0 (p.79)	R	The floating-point registers are now extended to hold either a single, double, or quad-precision floating-point value (FLEN=128).
RVU.13.7	13.0 (p.79)	R	The NaN-boxing scheme described in Section 12.2 is now extended recursively to allow a single-precision value to be NaN-boxed inside a double precision value which is itself NaN-boxed inside a quad-precision value.
RVU.13.8	13.1 (p.79)	H	Quad-Precision Load and Store Instructions
RVU.13.9	13.1 (p.79) & 24.0 (p.129, p.135)	R	FLQ Instruction Load a quad precision number from memory to register operation **Encoding**: I-Type



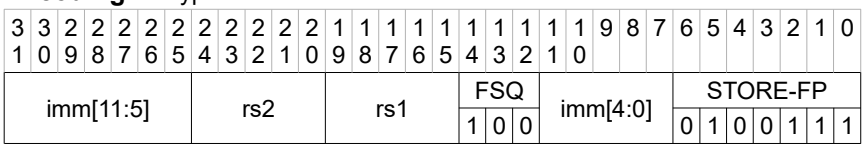
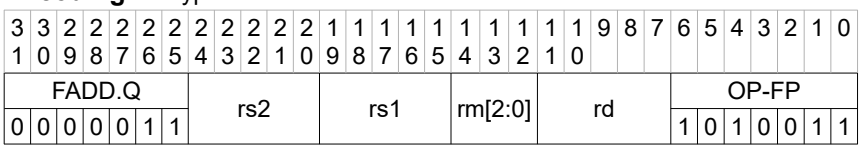
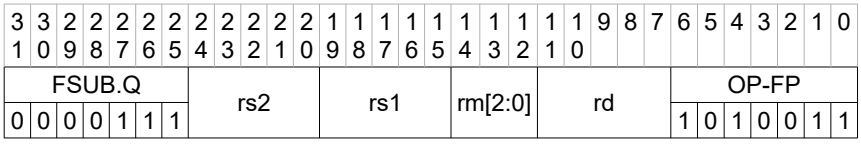
Valid Base: RV32, RV64

Task:
```
mem_addr=x(rs1)+I-Immediate
mem_val=MEM(mem_addr:mem_addr+15)
f(rd)=mem_val;
```

Explanation: The memory address is obtained by adding integer register *rs1* to the I-Immediate value. It loads one (128-bit) quad precision value from memory at the effective address and stores it into floating-point register *rd*.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.10	13.1 (p.79) & 24.0 (p.129, p.135)	R	FSQ Instruction Store quad precision number from register to memory operation **Encoding**: S-Type  **Valid Base**: RV32 RV64 **Task**: `mem_addr=x(rs1)+S-Immediate` `MEM(mem_addr:mem_addr+15)=f(rs2)[127:0]` **Explanation**: The memory address is obtained by adding integer register *rs1* to the S-Immediate value. It stores the contents of floating-point register *rs2* (128 bits) into the memory at the effective address. **Special Case**: none **Exception**: none
RVU.13.11	13.1 (p.79)	R	FLQ and FSQ are only guaranteed to execute atomically if the effective address is naturally aligned and XLEN=128.
RVU.13.12	13.1 (p.80)	R	FLQ and FSQ do not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.
RVU.13.13	13.2 (p.80)	H	Quad-Precision Computational Instructions
RVU.13.14	13.2 (p.80) & 24.0 (p.129, p.135)	R	FADD.Q Instruction Quad precision floating point addition operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)+f(rs2)` **Explanation**: FADD.Q performs the quad precision addition of *rs1* and *rs2* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none
RVU.13.15	13.2 (p.80) & 24.0 (p.129, p.135)	R	FSUB.Q Instruction Quad precision floating point subtraction operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)-f(rs2)` **Explanation**: FSUB.Q performs the quad precision subtraction of *rs2* from *rs1* and writes the result into floating point register *rd*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.16	13.2 (p.80) & 24.0 (p.129, p.135)	R	FMUL.Q Instruction Quad precision floating point multiplication operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMUL.Q							rs2					rs1			rm[2:0]		rd					OP-FP									
0	0	0	1	0	1	1																1	0	1	0	0	1	1			

Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)*f(rs2)`
Explanation: FMUL.Q performs the quad precision multiplication of *rs1* and *rs2* and writes the result into floating point register *rd*.
Special Case: none
Exception: none

RVU.13.17	13.2 (p.80) & 24.0 (p.129, p.135)	R	FDIV.Q Instruction Quad precision floating point division operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FDIV.Q							rs2					rs1			rm[2:0]		rd					OP-FP									
0	0	0	1	1	1	1																1	0	1	0	0	1	1			

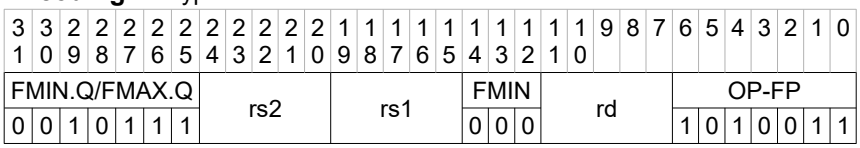
Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)/f(rs2)`
Explanation: FDIV.Q performs the quad precision division of *rs1* by *rs2* and writes the result into floating point register *rd*.
Special Case: none
Exception: none

RVU.13.18	13.2 (p.80) & 24.0 (p.129, p.135)	R	FSQRT.Q Instruction Quad precision floating point square root operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FSQRT.Q							rs2					rs1			rm[2:0]		rd					OP-FP									
0	1	0	1	1	1	1																1	0	1	0	0	1	1			

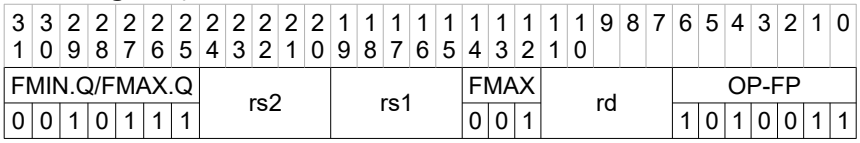
Valid Base: RV32, RV64
Task: `f(rd)=sqrt(f(rs1))`
Explanation: FSQRT.Q performs the quad precision square root of *rs1* and writes the result into floating point register *rd*. *rs2* value is expected to be zero.
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.19	13.2 (p.80) & 24.0 (p.129, p.135)	R	FMIN.Q Instruction Quad precision floating point minimum operation **Encoding**: R-Type



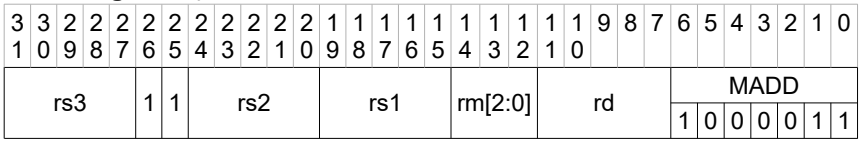
Valid Base: RV32, RV64
Task: `f(rd)=min(f(rs1),f(rs2))`
Explanation: FMIN.Q performs the quad precision minimum of *rs1* and *rs2* and writes the result into floating point register *rd*.
Special Case: -0.0 is considered smaller than +0.0
If both inputs are NaNs, the result is the canonical NaN.
If only one operand is a NaN, the result is the non-NaN operand.
Exception: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

RVU.13.20	13.2 (p.80) & 24.0 (p.129, p.135)	R	FMAX.Q Instruction Quad precision floating point maximum operation **Encoding**: R-Type

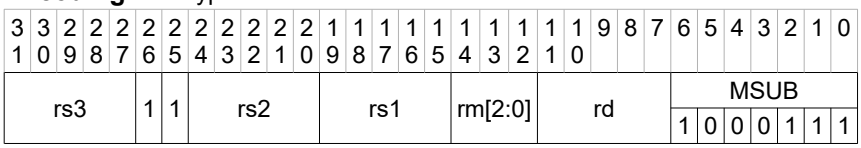
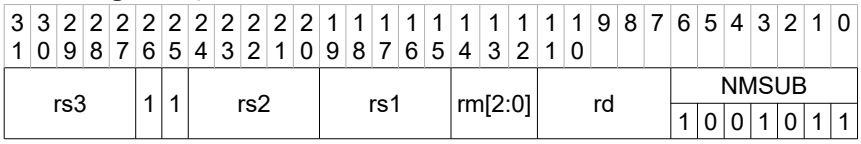
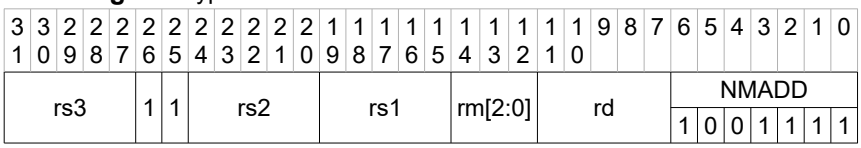


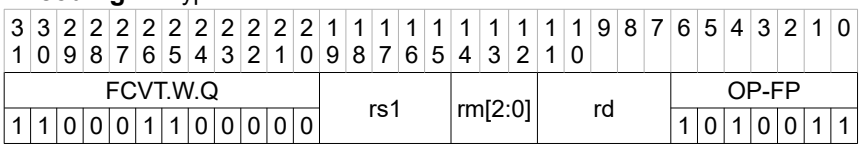
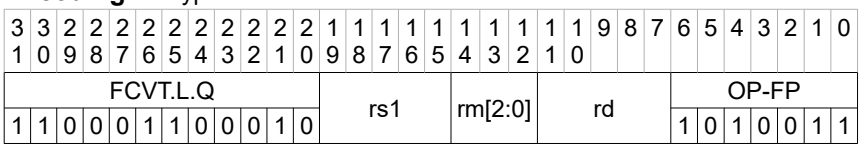
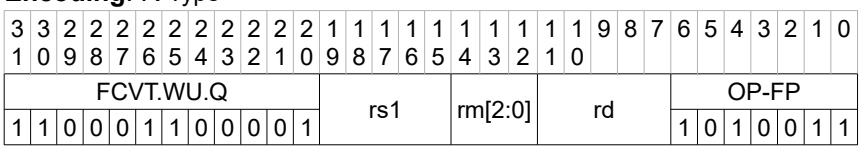
Valid Base: RV32, RV64
Task: `f(rd)=max(f(rs1),f(rs2))`
Explanation: FMAX.Q performs the quad precision maximum of *rs1* and *rs2* and writes the result into floating point register *rd*.
Special Case: -0.0 is considered smaller than +0.0
If both inputs are NaNs, the result is the canonical NaN.
If only one operand is a NaN, the result is the non-NaN operand.
Exception: Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

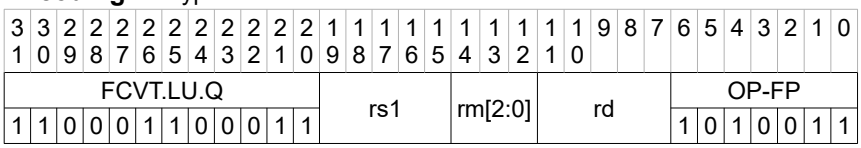
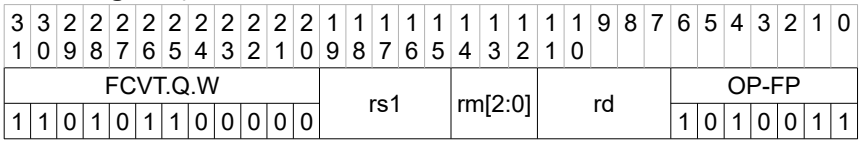
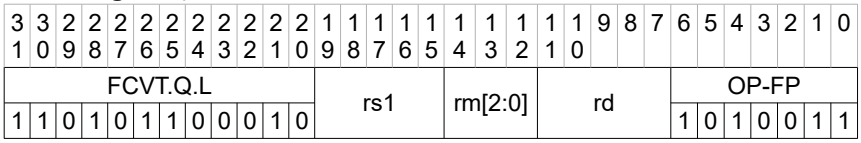
RVU.13.21	13.2 (p.80) & 24.0 (p.129, p.135)	R	FMADD.Q Instruction Quad precision multiply-add operation **Encoding**: R4-Type

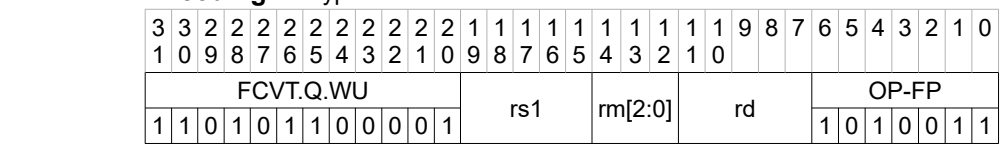
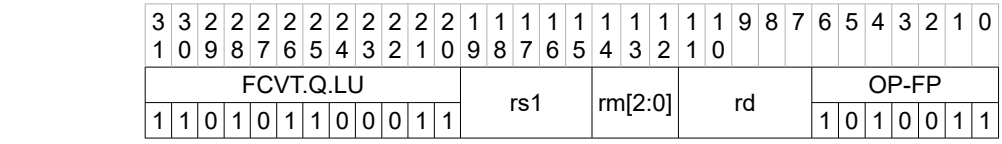
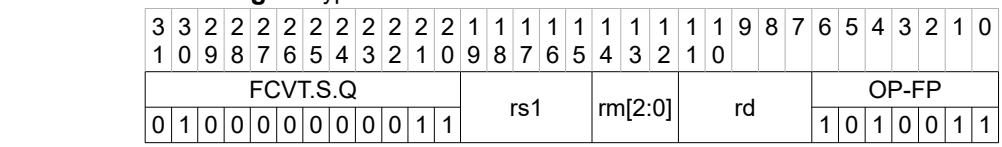


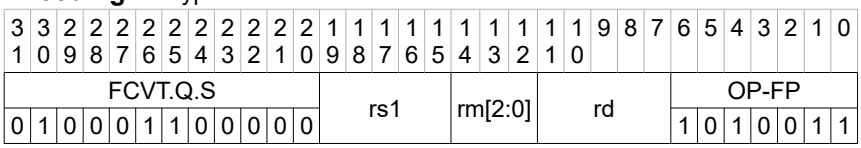
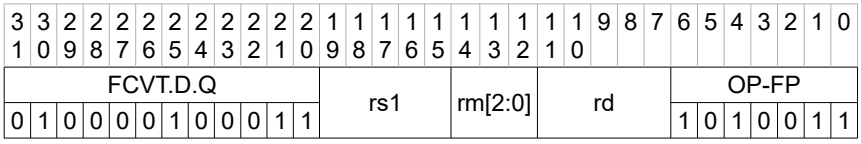
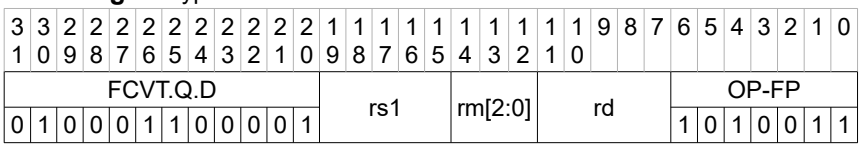
Valid Base: RV32, RV64
Task: `f(rd)=f(rs1)*f(rs2)+f(rs3)`
Explanation: FMADD.Q multiplies the values in *rs1* and *rs2*, adds the value in *rs3*, and writes the final result to *rd*.
Special Case: none
Exception: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.

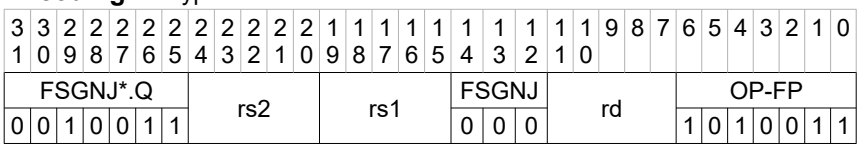
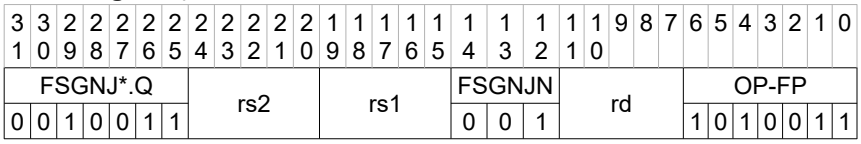
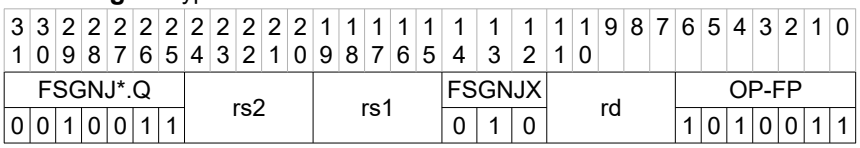
ID	REFERENCE	TYPE	DEFINITION
RVU.13.22	13.2 (p.80) & 24.0 (p.129, p.135)	R	FMSUB.Q Instruction Quad precision multiply-subtract operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=f(rs1)*f(rs2)-f(rs3)` **Explanation**: FMSUB.Q multiplies the values in *rs1* and *rs2*, subtracts the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.13.23	13.2 (p.80) & 24.0 (p.129, p.135)	R	FNMSUB.Q Instruction Quad precision multiply-subtract negate operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-(f(rs1)*f(rs2))+f(rs3)` **Explanation**: FNMSUB.Q multiplies the values in *rs1* and *rs2*, negates the product, adds the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.13.24	13.2 (p.80) & 24.0 (p.129, p.135)	R	FNMADD.Q Instruction Quad precision multiply-add negate operation **Encoding**: R4-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-(f(rs1)*f(rs2))-f(rs3)` **Explanation**: FNMADD.Q multiplies the values in *rs1* and *rs2*, negates the product, subtracts the value in *rs3*, and writes the final result to *rd*. **Special Case**: none **Exception**: The fused multiply-add instructions must set the invalid operation exception flag when the multiplicands are ∞ and zero, even when the addend is a quiet NaN.
RVU.13.25	13.3 (p.80)	H	Quad-Precision Convert and Move Instructions

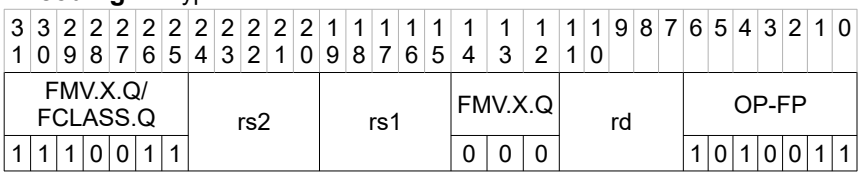
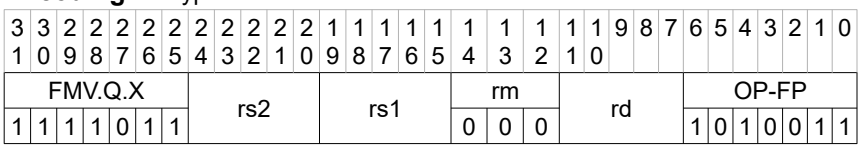
ID	REFERENCE	TYPE	DEFINITION
RVU.13.26	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.W.Q Instruction Quad precision floating point to 32-bit integer conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=Q2INT(f(rs1))` **Explanation**: FCVT.W.Q converts a quad precision floating-point number in floating point register *rs1* to a signed 32-bit integer in integer register *rd*. If XLEN>32, result is sign extended. f(*rs1*) is rounded first according to *rm* field. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. If f(*rs1*) < -2^{31}, x(*rd*) = -2^{31} If f(*rs1*) > 2^{31}-1 or NaN, x(*rd*) = 2^{31}-1 **Exception**: If f(*rs1*) < -2^{31}, invalid flag is set If f(*rs1*) > 2^{31}-1 or NaN, invalid flag is set
RVU.13.27	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.L.Q Instruction Quad precision floating point to 64-bit integer conversion operation **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `x(rd)=Q2INT(f(rs1))` **Explanation**: FCVT.L.Q converts a quad precision floating-point number in floating point register *rs1* to a signed 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. If f(*rs1*) < -2^{63}, x(*rd*) = -2^{63} If f(*rs1*) > 2^{63}-1 or NaN, x(*rd*) = 2^{63}-1 **Exception**: If f(*rs1*) < -2^{63}, invalid flag is set If f(*rs1*) > 2^{63}-1 or NaN, invalid flag is set
RVU.13.28	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.WU.Q Instruction Quad precision floating point to unsigned 32-bit integer conversion **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=Q2UINT(f(rs1))` **Explanation**: FCVT.WU.Q converts a quad precision floating-point number in floating point register *rs1* to an unsigned 32-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. If f(*rs1*) < 0, x(*rd*) = 0 If f(*rs1*) > 2^{31}-1 or NaN, x(*rd*) = 2^{31}-1 **Exception**: If f(*rs1*) < 0, invalid flag is set If f(*rs1*) > 2^{31}-1 or NaN, invalid flag is set

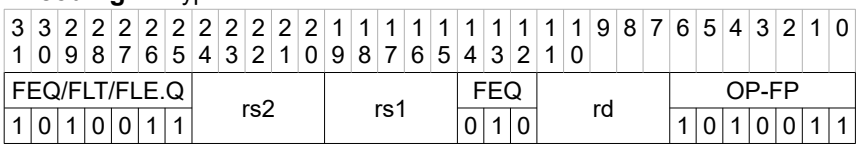
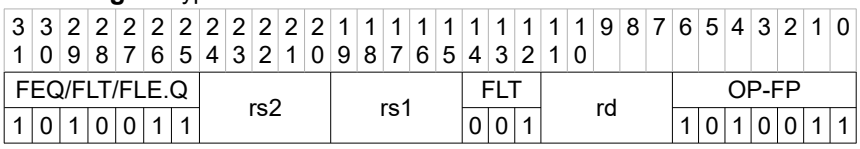
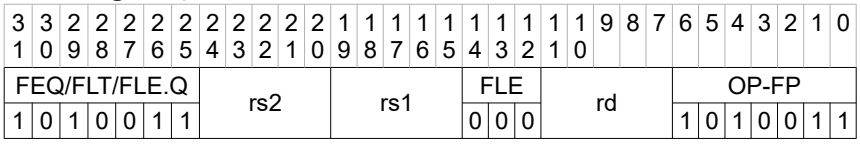
ID	REFERENCE	TYPE	DEFINITION
RVU.13.29	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.LU.Q Instruction Quad precision floating point to unsigned 64-bit integer conversion **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `x(rd)=Q2UINT(f(rs1))` **Explanation**: FCVT.LU.Q converts a quad precision floating-point number in floating point register *rs1* to an unsigned 64-bit integer in integer register *rd*. f(*rs1*) is rounded first according to *rm* field. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. If f(*rs1*) < 0, x(*rd*) = 0 If f(*rs1*) > 2^{63}-1 or NaN, x(*rd*) = 2^{63}-1 **Exception**: If f(*rs1*) < 0, invalid flag is set If f(*rs1*) > 2^{63}-1 or NaN, invalid flag is set
RVU.13.30	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.W Instruction 32-bit integer to quad precision floating point conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=INT2Q(x(rs1))` **Explanation**: FCVT.Q.W converts a 32 bit signed integer number in integer register *rs1* to a quad precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to rm field. **Special Case**: none **Exception**: none
RVU.13.31	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.L Instruction 64-bit integer to quad precision floating point conversion operation **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `f(rd)=INT2Q(x(rs1))` **Explanation**: FCVT.Q.L converts a 64-bit signed integer number in integer register *rs1* to a quad precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.32	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.WU Instruction 32-bit unsigned integer to quad precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=UINT2Q(x(rs1))` **Explanation**: FCVT.Q.WU converts a 32 bit unsigned integer number in integer register *rs1* to a quad-precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.13.33	13.3 (p.80, p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.LU Instruction 64-bit unsigned integer to quad precision floating point conversion **Encoding**: R-Type  **Valid Base**: RV64 **Task**: `f(rd)=UINT2D(x(rs1))` **Explanation**: FCVT.Q.LU converts a 64-bit unsigned integer number in integer register *rs1* to a quad-precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.13.34	13.3 (p.81) & 24.0 (p.129, p.135)	R	FCVT.S.Q Instruction Quad precision floating point to single precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=Q2F(f(rs1))` **Explanation**: FCVT.S.Q converts a quad precision floating point number in *rs1* to a single precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.35	13.3 (p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.S Instruction Single precision floating point to quad precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=F2Q(f(rs1))` **Explanation**: FCVT.Q.S converts a single precision floating point number in *rs1* to a quad precision floating-point number in floating point register *rd*. **Special Case**: none **Exception**: none
RVU.13.36	13.3 (p.81) & 24.0 (p.129, p.135)	R	FCVT.D.Q Instruction Quad precision floating point to double precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=Q2D(f(rs1))` **Explanation**: FCVT.D.Q converts a quad precision floating point number in *rs1* to a double precision floating-point number in floating point register *rd*. f(*rd*) is rounded according to *rm* field. **Special Case**: none **Exception**: none
RVU.13.37	13.3 (p.81) & 24.0 (p.129, p.135)	R	FCVT.Q.D Instruction Double precision floating point to quad precision conversion operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=D2Q(f(rs1))` **Explanation**: FCVT.Q.D converts a double precision floating point number in *rs1* to a quad precision floating-point number in floating point register *rd*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.38	13.3 (p.81) & 24.0 (p.129, p.135)	R	FSGNJ.Q Instruction Change sign to register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJ.Q changes the sign of the number in floating point register at *rs1* to the sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to sign bit of *rs2*. **Special Case**: none **Exception**: none
RVU.13.39	13.3 (p.81) & 24.0 (p.129, p.135)	R	FSGNJN.Q Instruction Change sign to opposite of register sign operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=-1*SGN(f(rs2))*|f(rs1)|` **Explanation**: FSGNJN.Q changes the sign of the number in floating point register at *rs1* to the opposite sign of the number in floating point register at *rs2* and stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the not of the sign bit of *rs2*. **Special Case**: none **Exception**: none
RVU.13.40	13.3 (p.81) & 24.0 (p.129, p.135)	R	FSGNJX.Q Instruction Change sign to multiplication of signs operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `f(rd)=SGN(f(rs2))*SGN(f(rs1))*|f(rs1)|` **Explanation**: FSGNJX.Q changes the sign of the number in floating point register at *rs1* to the multiplication of the sign of the number in floating point register at *rs1* and *rs2*, and then stores the result to floating point register *rd*. Effectively replace the sign bit of *rs1* to the xor of sign bit of *rs1* and sign bit of *rs2*. **Special Case**: none **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.13.41	13.3 (p.81) & 24.0 (p.129, p.135)	R	FMV.X.Q Instruction Move quad precision floating point register to integer register operation **Encoding**: R-Type  **Valid Base**: RV128 **Task**: `x(rd)[127:0]=f(rs1);` **Explanation**: FMV.X.Q moves the quad-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the 128 bit integer register *rd*. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. *rs2* is not used and expected to be zero. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. **Exception**: none
RVU.13.42	13.3 (p.81) & 24.0 (p.129, p.135)	R	FMV.Q.X Instruction Move integer register to quad precision floating point register operation **Encoding**: R-Type  **Valid Base**: RV128 **Task**: `f(rd)=x(rs1)[127:0];` **Explanation**: FMV.Q.X moves the quad-precision value encoded in IEEE 754-2008 standard encoding from 128 bits 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. *rs2* is not used and expected to be zero. **Special Case**: none **Exception**: none
RVU.13.43	13.3 (p.81)	I	FMV.X.Q and FMV.Q.X instructions are not provided in RV32 or RV64, so quad-precision bit patterns must be moved to the integer registers via memory.
RVU.13.44	13.3 (p.81)	C	RV128 will support FMV.X.Q and FMV.Q.X in the Q extension.
RVU.13.45	13.4 (p.81)	H	Quad-Precision Floating-Point Compare Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.13.46	13.4 (p.81) & 24.0 (p.129, p.135)	R	FEQ.Q Instruction Quad precision floating point equality check operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)=f(rs2); x(rd)=0 otherwise` **Explanation**: FEQ.Q performs the equality comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FEQ.Q performs a quiet comparison: it only sets the invalid operation exception flag if either input is a signaling NaN.
RVU.13.47	13.4 (p.81) & 24.0 (p.129, p.135)	R	FLT.Q Instruction Quad precision floating point less than comparison operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)<f(rs2); x(rd)=0 otherwise` **Explanation**: FLT.Q performs the less than comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FLT.Q performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.
RVU.13.48	13.4 (p.81) & 24.0 (p.129, p.135)	R	FLE.Q Instruction Quad precision floating point less than comparison operation **Encoding**: R-Type  **Valid Base**: RV32, RV64 **Task**: `x(rd)=1 when f(rs1)≤f(rs2); x(rd)=0 otherwise` **Explanation**: FLE.Q performs the less than and equal to comparison between floating-point registers *rs1* and *rs2* writing 1 to the integer register *rd* if the condition holds, and 0 otherwise. **Special Case**: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction. The result is 0 if either operand is NaN. **Exception**: FLE.Q performs what the IEEE 754-2008 standard refers to as signaling comparisons: that is, they set the invalid operation exception flag if either input is NaN.
RVU.13.49	13.5 (p.32)	H	Quad-Precision Floating-Point Classify Instruction

ID	REFERENCE	TYPE	DEFINITION
RVU.13.50	13.5 (p.82) & 24.0 (p.129, p.135) Table 11.5	R	FCLASS.Q Instruction Quad precision floating point classify operation **Encoding**: R-Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FMV.X.Q/ FCLASS.Q							rs2					rs1					FCLASS			rd					OP-FP						
1	1	1	0	0	1	1											0	0	1							1	0	1	0	0	1	1

Valid Base: RV32, RV64

Task: `x(rd)= CLASS(f(rs1))`

Explanation: The FCLASS.Q 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. *rs2* is not used and expected to be zero. The format of the mask is described in Table below. 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.

***rd* bit**	**Meaning**
0	Rs1 is −∞
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 +∞.
8	rs1 is a signaling NaN.
9	rs1 is a quiet NaN.

Special Case: If *rd* is zero (addressing x0), result is ignored or instruction is treated as a NOP instruction.

Exception: none

CHAPTER 14 RVWMO Memory Consistency Model

ID	REFERENCE	TYPE	DEFINITION
RVU.14.1	14.0 (p.83)	H	RVWMO Memory Consistency Model
RVU.14.2	14.0 (p.83) preface (p.i)	I	RVWMO version is 2.0 and status is ratified. (RVWMO version is given as 2.0 at the ratification table in preface, but listed as 0.1 at the chapter headings of both Chapter 14 & Appendix A)
RVU.14.3	14.0 (p.83)	I	This chapter defines the RISC-V memory consistency model.
RVU.14.4	14.0 (p.83)	I	A memory consistency model is a set of rules specifying the values that can be returned by loads of memory.
RVU.14.5	14.0 (p.83)	R	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.
RVU.14.6	14.0 (p.83)	R	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.
RVU.14.7	14.0 (p.83)	I	Therefore, multithreaded code may require explicit synchronization to guarantee ordering between memory instructions from different harts. The base RISC-V ISA provides a FENCE instruction for this purpose, described in Section 2.7, while the atomics extension “A” additionally defines load-reserved/store-conditional and atomic read-modify-write instructions.
RVU.14.8	14.0 (p.83)	I	The standard ISA extension for misaligned atomics “Zam” (Chapter 22) and the standard ISA extension for total store ordering “Ztso” (Chapter 23) augment RVWMO with additional rules specific to those extensions.
RVU.14.9	14.0 (p.83)	I	The appendices to this specification provide both axiomatic and operational formalizations of the memory consistency model as well as additional explanatory material.
RVU.14.10	14.0 (p.83)	C	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.
RVU.14.11	14.0 (p.83)	C	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.
RVU.14.12	14.1 (p.84)	H	Definition of the RVWMO Memory Model
RVU.14.13	14.1 (p.84)	I	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.14.14	14.1 (p.84)	I	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).
RVU.14.15	14.1 (p.84)	H	Memory Model Primitives
RVU.14.16	14.1 (p.84)	I	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.
RVU.14.17	14.1 (p.84)	I	Memory-accessing instructions give rise to *memory operations*. A memory operation can be either a *load operation*, a *store operation*, or both simultaneously.
RVU.14.18	14.1 (p.84)	R	All memory operations are single-copy atomic: they can never be observed in a partially-complete state.
RVU.14.19	14.1 (p.84)	R	Among instructions in RV32GC and RV64GC, each aligned memory instruction gives rise to exactly one memory operation, with two exceptions. • First, an unsuccessful SC instruction does not give rise to any memory operations. • Second, FLD and FSD instructions may each give rise to multiple memory operations if XLEN<64, as stated in Section 12.3 and clarified below
RVU.14.20	14.1 (p.84)	R	An aligned AMO gives rise to a single memory operation that is both a load operation and a store operation simultaneously.
RVU.14.21	14.1 (p.84)	C	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.
RVU.14.22	14.1 (p.84)	R	A misaligned load or store instruction may be decomposed into a set of component memory operations of any granularity.
RVU.14.23	14.1 (p.84)	R	An FLD or FSD instruction for which XLEN<64 may also be decomposed into a set of component memory operations of any granularity.
RVU.14.24	14.1 (p.84)	R	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.
RVU.14.25	14.1 (p.84)	R	The atomics extension "A" does not require execution environments to support misaligned atomic instructions at all; however, if misaligned atomics are supported via the "Zam" extension, LRs, SCs, and AMOs may be decomposed subject to the constraints of the atomicity axiom for misaligned atomics, which is defined in Chapter 22.
RVU.14.26	14.1 (p.84)	C	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.14.27	14.1 (p.85)	I	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).
RVU.14.28	14.1 (p.85)	I	The complete list of conditions determining whether an SC must succeed, may succeed, or must fail is defined in Section 8.2.
RVU.14.29	14.1 (p.85)	I	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”.
RVU.14.30	14.1 (p.85)	R	An AMO or LR instruction with *aq* set has an “acquire-RCsc” annotation.
RVU.14.31	14.1 (p.85)	R	An AMO or SC instruction with *rl* set has a “release-RCsc” annotation.
RVU.14.32	14.1 (p.85)	R	An AMO, LR, or SC instruction with both *aq* and *rl* set has both “acquire-Rcsc” and “release-RCsc” annotations.
RVU.14.33	14.1 (p.85)	I	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.
RVU.14.34	14.1 (p.85)	C	In the memory model literature, the term “RCpc” stands for release consistency with processorconsistent synchronization operations, and the term “RCsc” stands for release consistency with sequentially-consistent synchronization operations[‡‡] .
RVU.14.35	14.1 (p.85)	C	While there are many different definitions for acquire and release annotations in the literature, in the context of RVWMO these terms are concisely and completely defined by Preserved Program Order rules 5–7.
RVU.14.36	14.1 (p.85)	C	“RCpc” annotations are currently only used when implicitly assigned to every memory access per the standard extension “Ztso” (Chapter 23). Furthermore, although the ISA does not currently contain native load-acquire or store-release instructions, nor RCpc variants thereof, the RVWMO model itself is designed to be forwards-compatible with the potential addition of any or all of the above into the ISA in a future extension.
RVU.14.37	14.1 (p.85)	H	Syntactic Dependencies
RVU.14.38	14.1 (p.85)	I	The definition of the RVWMO memory model depends in part on the notion of a syntactic dependency, defined as follows.
RVU.14.39	14.1 (p.85)	I	In the context of defining dependencies, a “register” refers either to an entire general-purpose register, some portion of a CSR, or an entire CSR. The granularity at which dependencies are tracked through CSRs is specific to each CSR and is defined in Section 14.2.
RVU.14.40	14.1 (p.85)	I	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.

‡‡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.

ID	REFERENCE	TYPE	DEFINITION
RVU.14.41	14.1 (p.85, p.86)	R	In general, a register *r* other than x0 is a *source register* for an instruction *i* if any of the following hold: • In the opcode of *i*, *rs1*, *rs2*, or *rs3* is set to *r* • *i* is a CSR instruction, and in the opcode of *i*, csr is set to *r*, unless *i* is CSRRW or CSRRWI and *rd* is set to x0 • *r* is a CSR and an implicit source register for *i*, as defined in Section 14.3 • *r* is a CSR that aliases with another source register for i
RVU.14.42	14.1 (p.86)	I	Memory instructions also further specify which source registers are *address source registers* and which are *data source registers*.
RVU.14.43	14.1 (p.86)	R	In general, a register *r* other than x0 is a *destination register* for an instruction *i* if any of the following hold: • In the opcode of *i*, *rd* is set to *r* • *i* is a CSR instruction, and in the opcode of *i*, csr is set to *r*, unless *i* is CSRRS or CSRRC and *rs1* is set to x0 or *i* is CSRRSI or CSRRCI and *uimm[4:0]* is set to zero. • *r* is a CSR and an implicit destination register for *i*, as defined in Section 14.3 • *r* is a CSR that aliases with another destination register for *i*
RVU.14.44	14.1 (p.86)	I	Most non-memory instructions carry a *dependency* from each of their source registers to each of their destination registers. However, there are exceptions to this rule, see Section 14.3
RVU.14.45	14.1 (p.86)	R	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*
RVU.14.46	14.1 (p.86)	R	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*
RVU.14.47	14.1 (p.86)	R	(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 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*
RVU.14.48	14.1 (p.86)	R	(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 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*.

ID	REFERENCE	TYPE	DEFINITION
RVU.14.49	14.1 (p.87)	C	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.
RVU.14.50	14.1 (p.87)	H	Preserved Program Order
RVU.14.51	14.1 (p.87)	R	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*.
RVU.14.52	14.1 (p.87)	R	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 conditions (from 1 to 13) hold:
RVU.14.53	14.1 (p.87)	R	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*
RVU.14.54	14.1 (p.87)	R	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*
RVU.14.55	14.1 (p.87)	R	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*
RVU.14.56	14.1 (p.87)	R	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*
RVU.14.57	14.1 (p.88)	H	Memory Model Axioms
RVU.14.58	14.1 (p.88)	R	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*.

ID	REFERENCE	TYPE	DEFINITION
RVU.14.59	14.1 (p.88)	R	**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
RVU.14.60	14.1 (p.88)	R	**Atomicity Axiom** If *r* and *w* are paired load and store operations generated by aligned LR and SC instructions in a hart *h*, *s* is a store to byte *x*, and *r* returns a value written by *s*, then *s* must precede *w* in the global memory order, and there can be no store from a hart other than *h* to byte *x* following *s* and preceding *w* in the global memory order.
RVU.14.61	14.1 (p.88)	C	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.
RVU.14.62	14.1 (p.88)	R	**Progress Axiom** No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.
RVU.14.63	14.2 (p.88)	H	CSR Dependency Tracking Granularity
RVU.14.64	14.2 (p.88) Table 14.1		Granularities at which syntactic dependencies are tracked through CSRs

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

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

RVU.14.65	14.3 (p.88)	H	Source and Destination Register Listings
RVU.14.66	14.3 (p.88)	I	This section provides a concrete listing of the source and destination registers for each instruction. These listings are used in the definition of syntactic dependencies in Section 14.1.
RVU.14.67	14.3 (p.89)	I	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.
RVU.14.68	14.3 (p.89)	I	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

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.14.69 14.3 (p.90) I RV32I Base Integer Instruction Set:

Instruction	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	$rs1^{A}$, $rs2^{D}$		

RVU.14.70 14.3 (p.90) I RV32I Base Integer Instruction Set (Continued):

Instruction	Source Registers	Destination Registers	Accumulating CSRs
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	

ID	REFERENCE	TYPE	DEFINITION
RVU.14.71	14.3 (p.90, p.91)	I	RV32I Base Integer Instruction Set (Continued):

Instruction	Source Registers	Destination Registers	Accumulating CSRs
SRL	rs1, rs2	rd	
SRA	rs1, rs2	rd	
OR	rs1, rs2	rd	
AND	rs1, rs2	rd	
FENCE			
FENCE.I			
ECALL			
EBREAK			
CSRRW‡	rs1, csr∗	rd, csr	
CSRRS‡	rs1, csr	rd+, csr	
CSRRC‡	rs1, csr	rd+, csr	
CSRRWI‡	csr∗	rd, csr	
CSRRSI‡	csr	rd, csr#	
CSRRCI‡	csr	rd, csr#	

‡ carries a dependency from csr to rd
∗ unless rd=x0
+ unless rs1=x0
# unless uimm[4:0]=0

RVU.14.72	14.3 (p.91)	I	RV64I Base Integer Instruction Set :

Instruction	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	

RVU.14.73	14.3 (p.91)	I	RV32M Standard Extension :

Instruction	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	

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.14.74 14.3 (p.92) I RV64M Standard Extension :

Instruction	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	

RVU.14.75 14.3 (p.92) I RV32A Standard Extension :

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

* if successful

RVU.14.76 14.3 (p.92) I RV64A Standard Extension :

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

* if successful

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.14.77 14.3 (p.93) I RV32F Base Integer Instruction Set :

Instruction	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
FMSUB.S	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FNMSUB.S	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FNMADD.S	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FADD.S	rs1, rs2, frm*	rd	NV, OF, UF, NX
FSUB.S	rs1, rs2, frm*	rd	NV, OF, UF, NX
FMUL.S	rs1, rs2, frm*	rd	NV, OF, UF, NX
FDIV.S	rs1, rs2, frm*	rd	NV, DZ, OF, UF, NX
FSQRT.S	rs1, frm*	rd	NV, NX
FSGNJ.S	rs1, rs2	rd	
FSGNJN.S	rs1, rs2	rd	

* if rm=111

RVU.14.78 14.3 (p.93) I RV32F Base Integer Instruction Set (Continued) :

Instruction	Source Registers	Destination Registers	Accumulating CSRs
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
FCVT.WU.S	rs1, frm*	rd	NV, NX
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	NV
FCVT.S.WU	rs1, frm*	rd	NV
FMV.W.X	rs1	rd	

* if rm=111

RVU.14.79 14.3 (p.93) I RV64F Base Integer Instruction Set :

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

* if rm=111

ID **REFERENCE** **TYPE** **DEFINITION**

RVU.14.80 14.3 (p.94) I RV32D Base Integer Instruction Set :

Instruction	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
FMSUB.D	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FNMSUB.D	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FNMADD.D	rs1, rs2, rs3, frm*	rd	NV, OF, UF, NX
FADD.D	rs1, rs2, frm*	rd	NV, OF, UF, NX
FSUB.D	rs1, rs2, frm*	rd	NV, OF, UF, NX
FMUL.D	rs1, rs2, frm*	rd	NV, OF, UF, NX
FDIV.D	rs1, rs2, frm*	rd	NV, DZ, OF, UF, NX
FSQRT.D	rs1, frm*	rd	NV, NX
FSGNJ.D	rs1, rs2	rd	
FSGNJN.D	rs1, rs2	rd	

* if rm=111

RVU.14.81 14.3 (p.94) I RV32D Base Integer Instruction Set (Continued) :

Instruction	Source Registers	Destination Registers	Accumulating CSRs
FSGNJX.D	rs1, rs2	rd	
FMIN.D	rs1, rs2	rd	NV
FMAX.D	rs1, rs2	rd	NV
FCVT.S.D	rs1, frm*	rd	NX
FCVT.D.S	rs1, frm*	rd	NX
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
FCVT.WU.D	rs1, frm*	rd	NV, NX
FCVT.D.W	rs1	rd	
FCVT.D.WU	rs1	rd	

* if rm=111

RVU.14.82 14.3 (p.94) I RV64D Base Integer Instruction Set :

Instruction	Source Registers	Destination Registers	Accumulating CSRs
FCVT.L.D	rs1, frm*	rd	NV, NX
FCVT.LU.D	rs1, frm*	rd	NV, NX
FMV.X.D	rs1	rd	
FCVT.D.L	rs1, frm*	rd	NX
FCVT.D.LU	rs1, frm*	rd	NX
FMV.X.D	rs1	rd	

* if rm=111

CHAPTER 16 “C” Standard Extension for Compressed Instructions

ID	REFERENCE	TYPE	DEFINITION
RVU.16.1	16.0 (p.97)	H	“C” Standard Extension for Compressed Instructions
RVU.16.2	16.0 (p.97) preface (p.i)	I	“C” extension version is 2.0 and status is ratified.
RVU.16.3	16.0 (p.97)	I	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.
RVU.16.4	16.0 (p.97)	I	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.
RVU.16.5	16.0 (p.97)	I	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.
RVU.16.6	16.1 (p.97)	H	Overview
RVU.16.7	16.1 (p.97)	I	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.
RVU.16.8	16.1 (p.97)	R	The C extension is compatible with all other standard instruction extensions
RVU.16.9	161.0 (p.97)	R	The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions
RVU.16.10	16.1 (p.97)	R	When C extension is used, both 16 and 32 bit instructions now able to start on any 16-bit boundary (IALIGN=16).
RVU.16.11	16.1 (p.97)	R	With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions.
RVU.16.12	16.1 (p.97)	C	Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.
RVU.16.13	16.1 (p.97, p.98)	I	The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 16.4, a few opcodes are used for different purposes depending on base ISA width. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C.
RVU.16.14	16.1 (p.98)	R	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.15	16.1 (p.98)	I	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.
RVU.16.16	16.1 (p.98)	C	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.
RVU.16.17	16.1 (p.98)	I	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.
RVU.16.18	16.1 (p.98)	I	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.
RVU.16.19	16.1 (p.98)	C	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.
RVU.16.20	16.1 (p.98)	R	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.
RVU.16.21	16.1 (p.98, p.99)	C	Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch, 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 supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.
RVU.16.22	16.1 (p.99)	C	In 1963, CDC introduced the Cray-designed CDC 6600, 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.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.23	16.1 (p.99)	C	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.
RVU.16.24	16.1 (p.99)	C	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.
RVU.16.25	16.1 (p.99)	C	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.
RVU.16.26	16.1 (p.99)	C	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[§§].
RVU.16.27	16.2 (p.99)	H	Compressed Instruction Formats
RVU.16.28	16.2 (p.99)	R	There are 9 compressed instruction formats: CR, CI, CSS, CIW, CL, CS, CA, CB and CJ.
RVU.16.29	16.2 (p.100) Table 16.1	R	CR-Type instruction format (Register)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct4				rd/rs1					rs2					op	

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

ID	REFERENCE	TYPE	DEFINITION
RVU.16.30	16.2 (p.100) Table 16.1	R	CI-Type instruction format (Immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			imm	rd/rs1					imm					op	

RVU.16.31	16.2 (p.100) Table 16.1	R	CSS-Type instruction format (Stack-relative store)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			imm						rs2					op	

RVU.16.32	16.2 (p.99)	R	CR, CI, and CSS can use any of the 32 RVI registers
RVU.16.33	16.2 (p.100) Table 16.1	R	CIW-Type instruction format (Wide immediate)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			imm								rd’			op	

RVU.16.34	16.2 (p.100) Table 16.1	R	CL-Type instruction format (Load)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			imm			rs1’			imm		rd’			op	

RVU.16.35	16.2 (p.100) Table 16.1	R	CS-Type instruction format (Store)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			imm			rs1’			imm		rs2'			op	

RVU.16.36	16.2 (p.100) Table 16.1	R	CA-Type instruction format (Arithmetic)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct6						rd’/rs1’			funct2		rs2’			op	

RVU.16.37	16.2 (p.100) Table 16.1	R	CB-Type instruction format (Branch)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			offset			rs1’			offset					op	

RVU.16.38	16.2 (p.100) Table 16.1	R	CJ-Type instruction format (Jump)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
funct3			Jump target											op	

RVU.16.39	16.2 (p.99)	R	CIW, CL, CS, CA, and CB use 8 integer and floating point registers.
RVU.16.40	16.2 (p.100) Table 16.2	R	Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, CA, and CB formats.

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

RVU.16.41	16.2 (p.99)	I	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.
RVU.16.42	16.2 (p.99)	I	CIW supplies an 8-bit immediate for the ADDI4SPN instruction.
RVU.16.43	16.2 (p.100)	C	The RISC-V ABI was changed to make the frequently used registers map to registers x8–x15. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E base ISA, which only has 16 integer registers
RVU.16.44	16.2 (p.100)	R	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.45	16.2 (p.100)	C	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.
RVU.16.46	16.2 (p.100)	I	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.
RVU.16.47	16.2 (p.100)	I	When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding.
RVU.16.48	16.2 (p.100)	R	Where immediates are sign-extended, the sign-extension is always from bit 12.
RVU.16.49	16.2 (p.100)	I	Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate muxes required.
RVU.16.50	16.2 (p.100)	C	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.
RVU.16.51	16.2 (p.100)	R	For many RVC instructions, zero-valued immediates are disallowed and x0 is not a valid 5-bit register specifier.
RVU.16.52	16.2 (p.100)	I	These restrictions free up encoding space for other instructions requiring fewer operand bits.
RVU.16.53	16.3 (p.101)	H	Load and Store Instructions
RVU.16.54	16.3 (p.101)	R	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.
RVU.16.55	16.3 (p.101)	R	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.
RVU.16.56	16.3 (p.101)	H	Stack-Pointer-Based Loads and Stores

ID	REFERENCE	TYPE	DEFINITION
RVU.16.57	16.3 (p.101) 16.8 (p.113) Table 16.7	R	C.LWSP Instruction Load a 32-bit data from memory to register **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LWSP			imm	rd					imm					C2	
0	1	0												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[3]	inst[2]	inst[12]	inst[6]	inst[5]	inst[4]	0	0

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(2)+Offset
mem_val=MEM(mem_addr:mem_addr+3)
x(rd)=mem_val #sign Extension?
```

Explanation: 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; the code points with rd=x0 are reserved.

Special Case: If rd is zero (x0), instruction is not classified as C.LWSP

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.58	16.3 (p.101) 16.8 (p.113) Table 16.7	R	C.LDSP Instruction Load a 64-bit data from memory to register **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LDSP			imm	rd					imm					C2	
0	1	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[4]	inst[3]	inst[2]	inst[12]	inst[6]	inst[5]	0	0	0

Valid Base: RV64, RV128

Task:
```
mem_addr=x(2)+Offset
mem_val=MEM(mem_addr:mem_addr+7)
x(rd)=mem_val
```

Explanation: 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; the code points with rd=x0 are reserved.

Special Case: If rd is zero (x0), instruction is not classified as C.LDSP

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.59	16.3 (p.101) 16.8 (p.113) Table 16.7	R	C.LQSP Instruction Load a 128-bit data from memory to register **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LQSP			imm	rd					imm					C2	
0	0	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	inst[5]	inst[4]	inst[3]	inst[2]	inst[12]	inst[6]	0	0	0	0

Valid Base: RV128

Task:
```
mem_addr=x(2)+Offset
mem_val=MEM(mem_addr:mem_addr+15)
x(rd)=mem_val
```

Explanation: 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; the code points with rd=x0 are reserved.

Special Case: If rd is zero (x0), instruction is not classified as C.LQSP

Exception: none

RVU.16.60	16.3 (p.101) 16.8 (p.113) Table 16.7	R	C.FLWSP Instruction Load a 32-bit single precision data from memory to floating point register **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FLWSP			imm	rd					imm					C2	
0	1	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[3]	inst[2]	inst[12]	inst[6]	inst[5]	inst[4]	0	0

Valid Base: RV32

Task:
```
mem_addr=x(2)+Offset
mem_val=MEM(mem_addr:mem_addr+3)
f(rd)=mem_val
```

Explanation: 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)`.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.61	16.3 (p.101) 16.8 (p.113) Table 16.7	R	C.FLDSP Instruction Load a double precision number from memory to floating point register **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C. FLDSP			imm	rd					imm					C2	
0	0	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[4]	inst[3]	inst[2]	inst[12]	inst[6]	inst[5]	0	0	0

Valid Base: RV32, RV64

Task:
```
mem_addr=x(2)+Offset
mem_val=MEM(mem_addr:mem_addr+7)
f(rd)=mem_val
```

Explanation: 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)`.

Special Case: none

Exception: none

RVU.16.62	16.3 (p.102) 16.8 (p.113) Table 16.7	R	C.SWSP Instruction Store a 32-bit data from register to memory **Encoding**: CSS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SWSP			imm						rs2					C2	
1	1	0												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[8]	inst[7]	inst[12]	inst[11]	inst[10]	inst[9]	0	0

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(2)+Offset
MEM(mem_addr:mem_addr+3)=x(rs2)[31:0]
```

Explanation: 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)`.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.63	16.3 (p.102) 16.8 (p.113) Table 16.7	R	C.SDSP Instruction Store a 64-bit data from register to memory **Encoding**: CSS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SDSP			imm						rs2					C2	
1	1	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[9]	inst[8]	inst[7]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV64, RV128

Task:
```
mem_addr=x(2)+Offset
MEM(mem_addr:mem_addr+7)=x(rs2)[63:0]
```

Explanation: 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)`.

Special Case: none

Exception: none

RVU.16.64	16.3 (p.102) 16.8 (p.113) Table 16.7	R	C.SQSP Instruction Store a 128-bit data from register to memory **Encoding**: CSS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C. SQSP			imm						rs2					C2	
1	0	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	inst[10]	inst[9]	inst[8]	inst[7]	inst[12]	inst[11]	0	0	0	0

Valid Base: RV128

Task:
```
mem_addr=x(2)+Offset
MEM(mem_addr:mem_addr+15)=x(rs2)
```

Explanation: 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)`.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.65	16.3 (p.102) 16.8 (p.113) Table 16.7	R	C.FSWSP: Instruction Store a single precision number from floating point register to memory **Encoding**: CSS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FSWSP			imm						rs2					C2	
1	1	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[8]	inst[7]	inst[12]	inst[11]	inst[10]	inst[9]	0	0

Valid Base: RV32
Task: `mem_addr=x(2)+Offset`
`MEM(mem_addr:mem_addr+3)=f(rs2)`
Explanation: 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).`
Special Case: none
Exception: none

RVU.16.66	16.3 (p.102) 16.8 (p.113) Table 16.7	R	C.FSDSP Instruction Store a double precision number from floating point register to memory **Encoding**: CSS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C. FSDSP			imm						rs2					C2	
1	0	1												1	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[9]	inst[8]	inst[7]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV32, RV64
Task: `mem_addr=x(2)+Offset`
`MEM(mem_addr:mem_addr+7)=f(rs2)`
Explanation: 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).`
Special Case: none
Exception: none

RVU.16.67	16.3 (p.102)	C	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.68	16.3 (p.102, p.103)	C	A common mechanism used in other ISAs to further reduce save/restore code size is loadmultiple and store-multiple instructions. We considered adopting these for RISC-V but noted the following drawbacks to these instructions: • These instructions complicate processor implementations. • For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions. • Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple. • Unlike the rest of the RVC instructions, the compiler would have to be aware of these instructions to both generate the instructions and to allocate registers in an order to maximize the chances of the them being saved and stored, since they would be saved and restored in sequential order. • Simple microarchitectural implementations will constrain how other instructions can be scheduled around the load and store multiple instructions, leading to a potential performance loss. • The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, CA, and CB formats. Furthermore, much of the gains can be realized in software by replacing prologue and epilogue code with subroutine calls to common prologue and epilogue code, a technique described in Section 5.6 of [***]. 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.
RVU.16.69	16.3 (p.103)	H	Register-Based Loads and Stores
RVU.16.70	16.3 (p.103) 16.8 (p.112) Table 16.5	R	C.LW Instruction Load a 32-bit data from memory to register **Encoding**: CL-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LW			imm			rs1’			imm		rd’			C0	
0	1	0												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	inst[5]	inst[12]	inst[11]	inst[10]	inst[6]	0	0

Valid Base: RV32, RV64, RV128

Task:
```
mem_addr=x(rs1’)+Offset
mem_val=MEM(mem_addr:mem_addr+3)
x(rd’)=mem_val
```

Explanation: 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')`.

Special Case: none

Exception: none

***AndrewWaterman. Design of the RISC-V Instruction Set Architecture. PhD thesis, University of California, Berkeley, 2016.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.71	16.3 (p.103) 16.8 (p.112) Table 16.5	R	C.LD Instruction Load a 64-bit data from memory to register **Encoding**: CL-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LD			imm			rs1’			imm		rd’			C0	
0	1	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[6]	inst[5]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV64, RV128

Task:
```
mem_addr=x(rs1’)+Offset
mem_val=MEM(mem_addr:mem_addr+7)
x(rd’)=mem_val
```

Explanation: 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')`

Special Case: none

Exception: none

RVU.16.72	16.3 (p.103) 16.8 (p.112) Table 16.5	R	C.LQ Instruction Load a 128-bit data from memory to register **Encoding**: CL-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LQ			imm			rs1’			imm		rd’			C0	
0	0	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[10]	inst[6]	inst[5]	inst[12]	inst[11]	0	0	0	0

Valid Base: RV128

Task:
```
mem_addr=x(rs1’)+Offset
mem_val=MEM(mem_addr:mem_addr+15)
x(rd’)=mem_val
```

Explanation: 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')`.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.73	16.3 (p.103) 16.8 (p.112) Table 16.5	R	C.FLW Instruction Load a single precision number from memory to floating point register **Encoding**: CL-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FLW			imm			rs1’			imm		rd’			C0	
0	1	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	inst[5]	inst[12]	inst[11]	inst[10]	inst[6]	0	0

Valid Base: RV32

Task:
```
mem_addr=x(rs1’)+Offset
mem_val=MEM(mem_addr:mem_addr+3)
f(rd’)=mem_val
```

Explanation: 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')`.

Special Case: none

Exception: none

RVU.16.74	16.3 (p.103) 16.8 (p.112) Table 16.5	R	C.FLD Instruction Load a double precision number from memory to floating point register **Encoding**: CL-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FLD			imm			rs1’			imm		rd’			C0	
0	0	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[6]	inst[5]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV32, RV64

Task:
```
mem_addr=x(rs1’)+Offset
mem_val=MEM(mem_addr:mem_addr+7)
f(rd’)=mem_val
```

Explanation: 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')`.

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.75	16.3 (p.104) 16.8 (p.112) Table 16.5	R	C.SW: Store a 32-bit data from register to memory **Encoding**: CS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SW			imm			rs1’			imm		rs2'			C0	
1	1	0												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	inst[5]	inst[12]	inst[11]	inst[10]	inst[6]	0	0

Valid Base: RV32, RV64, RV128
Task: `mem_addr=x(rs1’)+Offset`
`MEM(mem_addr:mem_addr+3)=x(rs2’)`
Explanation: 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').`
Special Case: none
Exception: none

RVU.16.76	16.3 (p.104) 16.8 (p.112) Table 16.5	R	C.SD Instruction Store a 64-bit data from register to memory **Encoding**: CS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SD			imm			rs1’			imm		rs2'			C0	
1	1	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[6]	inst[5]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV64, RV128
Task: `mem_addr=x(rs1’)+Offset`
`MEM(mem_addr:mem_addr+7)=x(rs2’)`
Explanation: 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').`
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.77	16.3 (p.104) 16.8 (p.112) Table 16.5	R	C.SQ Instruction Store a 128-bit data from register to memory **Encoding**: CS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SQ			imm			rs1’			imm		rs2'			C0	
1	0	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	inst[10]	inst[6]	inst[5]	inst[12]	inst[11]	0	0	0	0

Valid Base: RV128

Task:
```
mem_addr=x(rs1’)+Offset
MEM(mem_addr:mem_addr+15)=x(rs2’)
```

Explanation: 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').`

Special Case: none

Exception: none

RVU.16.78	16.3 (p.104) 16.8 (p.112) Table 16.5	R	C.FSW Instruction Store a single precision number from floating point register to memory **Encoding**: CS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FSW			imm			rs1’			imm		rs2'			C0	
1	1	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	inst[5]	inst[12]	inst[11]	inst[10]	inst[6]	0	0

Valid Base: RV32

Task:
```
mem_addr=x(rs1’)+Offset
MEM(mem_addr:mem_addr+3)=f(rs2’)
```

Explanation: 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').

Special Case: none

Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.79	16.3 (p.104) 16.8 (p.112) Table 16.5	R	C.FSD Instruction Store a double precision data from floating point register to memory **Encoding**: CS-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.FSD			imm			rs1’			imm		rs2'			C0	
1	0	1												0	0

Offset:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	inst[6]	inst[5]	inst[12]	inst[11]	inst[10]	0	0	0

Valid Base: RV32, RV64
Task: `mem_addr=x(rs1’)+Offset`
`MEM(mem_addr:mem_addr+7)=f(rs2’)`
Explanation: 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').`
Special Case: none
Exception: none

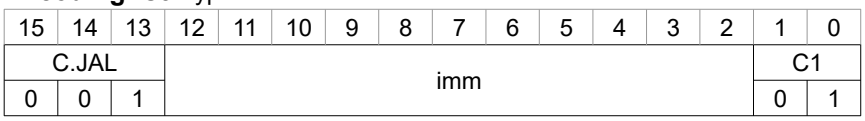
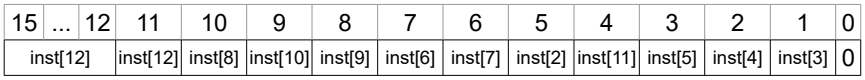
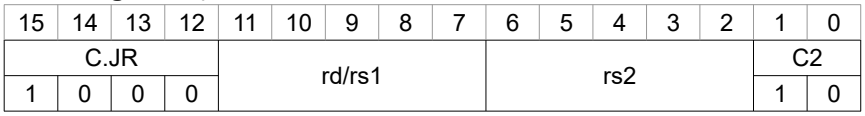
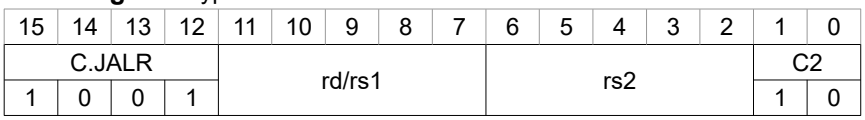
RVU.16.80	16.4 (p.104)	H	Control Transfer Instructions
RVU.16.81	16.4 (p.104)	I	RVC provides unconditional jump instructions and conditional branch instructions.
RVU.16.82	16.4 (p.104)	R	As with base RVI instructions, the offsets of all RVC control transfer instruction are in multiples of 2 bytes.
RVU.16.83	16.4 (p.104 p.105) 16.8 (p.113) Table 16.6	R	C.J Instruction Unconditional jump **Encoding**: CJ-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.J			imm											C1	
1	0	1												0	1

Offset:

15	...	12	11	10	9	8	7	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[8]	inst[10]	inst[9]	inst[6]	inst[7]	inst[2]	inst[11]	inst[5]	inst[4]	inst[3]	0

Valid Base: RV32, RV64, RV128
Task: `PC=PC+Offset`
Explanation: C.J performs an unconditional control transfer. The offset is sign-extended to XLEN bit 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].`
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.84	16.4 (p.104 p.105) 16.8 (p.113) Table 16.6	R	C.JAL Instruction Unconditional jump and link **Encoding**: CJ-Type  **Offset**:  **Valid Base**: RV32 **Task**: `x(1)=PC+2` `PC=PC+Offset` **Explanation**: 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]`. **Special Case**: none **Exception**: none
RVU.16.85	16.4 (p.105) 16.8 (p.113) Table 16.7	R	C.JR Instruction Unconditional register jump **Encoding**: CR-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `PC=x(rs1)` **Explanation**: C.JR (jump register) performs an unconditional control transfer to the address in register *rs1*. C.JR expands to `jalr x0, 0(rs1)`. C.JR is only valid when *rs1*≠x0; the code point with *rs1*=x0 is reserved. *rs2* is expected to be zero. **Special Case**: If rs1 is zero (x0), instruction is not classified as C.JR **Exception**: none
RVU.16.86	16.4 (p.105) 16.8 (p.113) Table 16.7	R	C.JALR Instruction Unconditional register jump and link **Encoding**: CR-Type  **Valid Base**: RV32, RV64, RV128 **Task**: `x(1)=PC+2` `PC=x(rs1)` **Explanation**: C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (`pc`+2) to the link register, `x1`. C.JALR expands to `jalr x1, 0(rs1). C.JALR` is only valid when rs1≠x0 **Special Case**: If rs1 is zero (x0), instruction is not classified as C.JALR **Exception**: none
RVU.16.87	16.4 (p.105)	C	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.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.88	16.4 (p.105) 16.8 (p.113) Table 16.6	R	C.BEQZ Instruction Branch if zero **Encoding**: CB-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.BEQZ			imm			rs1’			imm					C1	
1	1	0												0	1

Offset:

XLEN-1	...	9	8	7	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[2]	inst[11]	inst[10]	inst[4]	inst[3]	0

Valid Base: RV32, RV64, RV128
Task: `PC=PC+Offset if x(rs1’)=0; PC=PC+2 otherwise`
Explanation: C.BEQZ performs conditional control transfers. The offset is sign-extended to XLEN bit and added to the pc to form the branch target address. It can therefore target a ±256B range. C.BEQZ takes the branch if the value in register *rs1 ’* is zero. It expands to `beq rs1', x0, offset[8:1]`.
Special Case: none
Exception: none

RVU.16.89	16.4 (p.105) 16.8 (p.113) Table 16.6	R	C.BNEZ Instruction Branch if not zero **Encoding**: CB-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.BNEZ			imm			rs1’			imm					C1	
1	1	1												0	1

Offset:

XLEN-1	...	9	8	7	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[2]	inst[11]	inst[10]	inst[4]	inst[3]	0

Valid Base: RV32, RV64, RV128
Task: `PC=PC+Offset if x(rs1’)≠0; PC=PC+2 otherwise`
Explanation: C.BNEZ is defined analogously to C.BEQZ, but it takes the branch if *rs1’* contains a nonzero value. It expands to `bne rs1', x0, offset[8:1]`.
Special Case: none
Exception: none

RVU.16.90	16.5 (p.106)	H	Integer Computational Instructions
RVU.16.91	16.5 (p.106)	I	RVC provides several instructions for integer arithmetic and constant generation.
RVU.16.92	16.5 (p.106)	H	Integer Constant-Generation Instructions
RVU.16.93	16.5 (p.106)	I	The two constant-generation instructions both use the CI instruction format and can target any integer register.

ID	REFERENCE	TYPE	DEFINITION
RVU.16.94	16.5 (p.106) 16.8 (p.113) Table 16.6	R	C.LI Instruction Load immediate **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LI			imm	rd					imm					C1	
0	1	0												0	1

Immediate:

XLEN-1	...	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

Valid Base: RV32, RV64, RV128
Task: `x(rd)=Immediate`
Explanation: C.LI loads the sign-extended 6-bit Immediate, into register *rd*. C.LI expands into `addi rd, x0, imm[5:0]`. C.LI is only valid when rd≠x0; the code points with rd=x0 encode HINTs.
Special Case: If rd is zero (x0), it is considered as HINT
Exception: none

RVU.16.95	16.5 (p.106) 16.8 (p.113) Table 16.6	R	C.LUI Instruction Load upper immediate **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.LUI			imm	rd					imm					C1	
0	1	1												0	1

Immediate:

XLEN-1	...	18	17	16	15	14	13	12	11	...	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]	0		

Valid Base: RV32, RV64, RV128
Task: `x(rd)=Immediate`
Explanation: C.LUI loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI expands into `lui rd, nzimm[17:12]`. C.LUI is only valid when *rd*≠{x0, x2}, and when the immediate is not equal to zero. The code points with imm=0 are reserved; the remaining code points with *rd*=x0 are HINTs; and the remaining code points with *rd*=x2 correspond to the C.ADDI16SP instruction.
Special Case: If rd is zero (x0), it is not a classified as C.LUI but HINT.
If *rd* is two (x2), it is not classified as C.LUI but . C.ADDI16SP
If Imm is zero, it is not classified as C.LUI but reserved.
Exception: none

RVU.16.96	16.5 (p.106)	H	Integer Register-Immediate Operations

ID	REFERENCE	TYPE	DEFINITION
RVU.16.97	16.5 (p.106) 16.8 (p.113) Table 16.6	R	C.ADDI Instruction Add immediate **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.ADDI			imm	rd					imm					C1	
0	0	0												0	1

Immediate:

XLEN-1	...	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

Valid Base: RV32, RV64, RV128
Task: `x(rd)=x(rd)+Immediate`
Explanation: 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 and *imm*≠0. The code points with *rd*=x0 encode the C.NOP instruction; the remaining code points with imm=0 encode HINTs.
Special Case: If *rd* is zero (x0), it is not classified as C.ADDI but C.NOP. If *imm* is zero, it is not classified as C.ADDI but reserved.
Exception: none

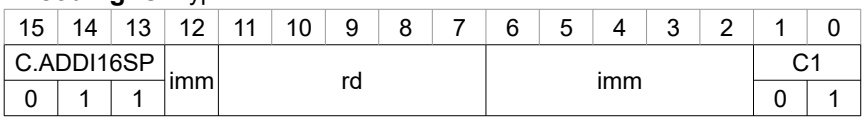
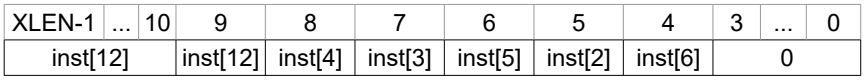
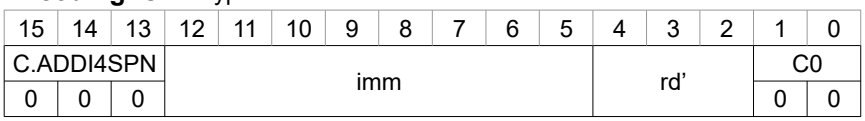
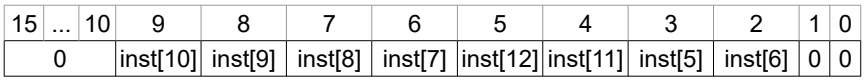
RVU.16.98	16.5 (p.106) 16.8 (p.113) Table 16.6	R	C.ADDIW Instruction Add immediate **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.ADDIW			imm	rd					imm					C1	
0	0	1												0	1

Immediate:

XLEN-1	...	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

Valid Base: RV64, RV128
Task: `x(rd)=x(rd)+Immediate`
Explanation: 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; the code points with *rd*=x0 are reserved.
Special Case: If *rd* is zero (x0), it is not classified as C.ADDIW but reserved.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.99	16.5 (p.106, p.107) 16.8 (p.113) Table 16.6	R	C.ADDI16SP Instruction Add immediate to stack pointer **Encoding**: CI-Type  **Offset**:  **Valid Base**: RV32, RV64, RV128 **Task**: `x(2)=x(2)+Offset` **Explanation**: C.ADDI16SP shares the opcode with C.LUI, but has a destination field of x2 (*rd*=2). C.ADDI16SP adds the non-zero sign-extended 6-bit offset to the value in the stack pointer (*sp*=x2), where the offset is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into `addi x2, x2, nzimm[9:4]`. C.ADDI16SP is only valid when imm≠0; the code point with imm=0 is reserved **Special Case**: If rd is NOT two (x2), it is not classified as C.ADDI16SP but C.LUI.. **Exception**: none
RVU.16.100	16.5 (p.107)	C	In the standard RISC-V calling convention, the stack pointer sp is always 16-byte aligned.
RVU.16.101	16.5 (p.107) 16.8 (p.112) Table 16.5	R	C.ADDI4SPN Instruction Add unsigned immediate to stack pointer and store **Encoding**: CIW-Type  **Offset**:  **Valid Base**: RV32, RV64, RV128 **Task**: `x(rd')=x(2)+Offset` **Explanation**: C.ADDI4SPN is a CIW-format instruction that adds a zero-extended non-zero offset, scaled by 4, to the stack pointer, *x2*, and writes the result to *rd'*. This instruction is used to generate pointers to stack-allocated variables, and expands to `addi rd ', x2, nzuimm[9:2]`. C.ADDI4SPN is only valid when *imm*≠0; the code points with *imm*=0 are reserved. **Special Case**: If *imm* is zero, it is not classified as C.ADDI4SPN but reserved **Exception**: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.102	16.5 (p.107) 16.8 (p.113) Table 16.7	R	C.SLLI Instruction Logical left shift **Encoding**: CI-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SLLI			imm	rd					imm					C2	
0	0	0												1	0

Immediate:

7	6	5	4	3	2	1	0
0	0	inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

Valid Base: RV32, RV64, RV128
Task: `x(rd)=x(rd)<<Immediate[5:0]`
Explanation: 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 immediate field. For RV128C, a shift amount of zero is used to encode a shift of 64. C.SLLI expands into `slli rd, rd, Immediate[5:0]`, except for RV128C with Immediate=0, which expands to `slli rd, rd, 64`.
Special Case: For RV32C, Immediate[5] must be zero; the code points with imm[5]=1 are reserved for custom extensions.
For RV32C and RV64C, the shift amount must be non-zero; the code points with imm=0 are HINTs.
For all base ISAs, the code points with *rd*=x0 are HINTs, except those with Immediate[5]=1 in RV32C.
Exception: none

RVU.16.103	16.5 (p.107) 16.8 (p.113) Table 16.6	R	C.SRLI Instruction Logical right shift **Encoding**: CB-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SRLI			imm	C.SRLI		rd’			imm					C1	
1	0	0		0	0									0	1

Immediate:

7	6	5	4	3	2	1	0
inst[12]*	inst[12]*	inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

* RV128 only
Valid Base: RV32, RV64, RV128
Task: `x(rd’)=x(rd’)>>Immediate[5:0]`
Explanation: 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 immediate field. For RV128C, a shift amount of zero is used to encode a shift of 64. Furthermore, the shift amount is sign-extended for RV128C, and so the legal shift amounts are 1–31, 64, and 96–127. C.SRLI expands into `srli rd', rd', Immediate[5:0]`, except for RV128C with Immediate=0, which expands to `srli rd', rd', 64`.
Special Case: For RV32C, Immediate[5] must be zero; the code points with Immediate[5]=1 are reserved for custom extensions. For RV32C and RV64C, the shift amount must be non-zero; the code points with Immediate=0 are HINTs.
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.104	16.5 (p.107, p.108) 16.8 (p.113) Table 16.6	R	C.SRAI Instruction Aritmetic right shift **Encoding**: CB-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SRAI			imm	C.SRAI		rd’			imm					C1	
1	0	0		0	1									0	1

Immediate:

7	6	5	4	3	2	1	0
inst[12]*	inst[12]*	inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

* RV128 only

Valid Base: RV32, RV64, RV128

Task: `x(rd’)=x(rd’)>>>Immediate[5:0]`

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

Special Case: For RV32C, Immediate[5] must be zero; the code points with Immediate[5]=1 are reserved for custom extensions. For RV32C and RV64C, the shift amount must be non-zero; the code points with Immediate=0 are HINTs.

Exception: none

RVU.16.105	16.5 (p.108)	C	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.
RVU.16.106	16.5 (p.108) 16.8 (p.113) Table 16.6	R	C.ANDI Instruction And register with immediate **Encoding**: CB-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.ANDI			imm	C.ANDI		rd’			imm					C1	
1	0	0		1	0									0	1

Immediate:

XLEN-1	...	6	5	4	3	2	1	0
inst[12]			inst[12]	inst[6]	inst[5]	inst[4]	inst[3]	inst[2]

Valid Base: RV32, RV64, RV128

Task: `x(rd’)=x(rd’) and Immediate`

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

Special Case: none

Exception: none

RVU.16.107	16.5 (p.108)	H	Integer Register-Register Operations

ID	REFERENCE	TYPE	DEFINITION
RVU.16.108	16.5 (p.108) 16.8 (p.113) Table 16.7	R	C.MV Instruction Move register **Encoding**: CR-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.MV				rd					rs2					C2	
1	0	0	0											1	0

Valid Base: RV32, RV64, RV128
Task: `x(rd)=x(rs2)`
Explanation: C.MV copies the value in register *rs2* into register *rd*. C.MV expands into `add rd, x0, rs2`. C.MV is only valid when *rs2*≠x0; the code points with *rs2*=x0 correspond to the C.JR instruction.
Special Case: The code points with *rs2*≠x0 and *rd*=x0 are HINTs.
Exception: none

RVU.16.109	16.5 (p.108)	C	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.
RVU.16.110	16.5 (p.108) 16.8 (p.113) Table 16.7	R	C.ADD Instruction Add register **Encoding**: CR-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.ADD				rd/rs1					rs2					C2	
1	0	0	1											1	0

Valid Base: RV32, RV64, RV128
Task: `x(rd)=x(rd)+x(rs2)`
Explanation: C.ADD adds the values in registers *rd* and *rs2* and writes the result to register *rd*. C.ADD expands into `add rd, rd, rs2`. C.ADD is only valid when *rs2*≠x0; the code points with *rs2*=x0 correspond to the C.JALR and C.EBREAK instructions. The code points with *rs2*≠x0 and rd=x0 are HINTs.
Special Case: The code points with *rs2*≠x0 and *rd*=x0 are HINTs.
Exception: none

RVU.16.111	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.AND Instruction And register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.AND						rd’			C.AND		rs2’			C1	
1	0	0	0	1	1				1	1				0	1

Valid Base: RV32, RV64, RV128
Task: `x(rd’)=x(rd’) and x(rs2’)`
Explanation: 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'`.
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.112	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.OR Instruction Or register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.OR						rd’			C.OR		rs2’			C1	
1	0	0	0	1	1				1	0				0	1

Valid Base: RV32, RV64, RV128
Task: `x(rd')=x(rd') or x(rs2')`
Explanation: 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'`.
Special Case: none
Exception: none

RVU.16.113	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.XOR Instruction Xor register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.XOR						rd’			C.XOR		rs2’			C1	
1	0	0	0	1	1				0	1				0	1

Valid Base: RV32, RV64, RV128
Task: `x(rd')=x(rd') xor x(rs2')`
Explanation: 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'`.
Special Case: none
Exception: none

RVU.16.114	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.SUB Instruction Subtract register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SUB						rd’			C.SUB		rs2’			C1	
1	0	0	0	1	1				0	0				0	1

Valid Base: RV32, RV64, RV128
Task: `x(rd')=x(rd')-x(rs2')`
Explanation: 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'`.
Special Case: none
Exception: none

ID	REFERENCE	TYPE	DEFINITION
RVU.16.115	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.ADDW Instruction Add 32 bit register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.ADDW						rd’			C.ADDW		rs2’			C1	
1	0	0	1	1	1				0	1				0	1

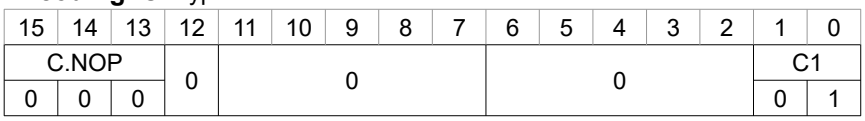
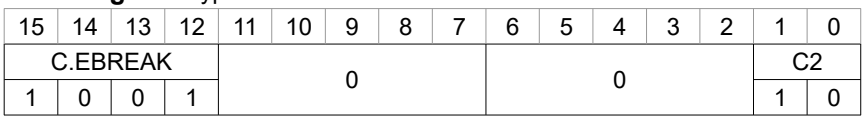
Valid Base: RV64, RV128
Task:
```
sum=x(rd’)+x(rs2’)
x(rd’)[31:0]=sum[31:0];
x(rd’)[XLEN-1:32]=sum[31]
```
Explanation: C.ADDW is an RV64C/RV128C-only instruction that adds the values in registers *rd′* and *rs2′*, then sign-extends the lower 32 bits of the sum before writing the result to register *rd′*. C.ADDW expands into `addw rd', rd', rs2'`.
Special Case: none
Exception: none

RVU.16.116	16.5 (p.109) 16.8 (p.113) Table 16.6	R	C.SUBW Instruction Subtract 32 bit register **Encoding:** CA-Type

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C.SUBW						rd’			C.SUBW		rs2’			C1	
1	0	0	1	1	1				0	0				0	1

Valid Base: RV64, RV128
Task:
```
sub=x(rd’)-x(rs2’)
x(rd’)[31:0]=sub[31:0];
x(rd’)[XLEN-1:32]=sub[31]
```
Explanation: 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'`.
Special Case: none
Exception: none

RVU.16.117	16.5 (p.109)	C	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.
RVU.16.118	16.5 (p.109)	H	Defined Illegal Instruction
RVU.16.119	16.5 (p.109)	R	A 16-bit instruction with all bits zero is permanently reserved as an illegal instruction.
RVU.16.120	16.5 (p.109, p.110)	C	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.
RVU.16.121	16.5 (p.110)	H	NOP Instruction

ID	REFERENCE	TYPE	DEFINITION
RVU.16.122	16.5 (p.110) 16.8 (p.113) Table 16.6	R	C.NOP Instruction No Operation **Encoding**: CI-Type  **Valid Base**: RV32, RV64, RV128 **Task**: – **Explanation**: C.NOP is a CI-format instruction that does not change any user-visible state, except for advancing the `pc` and incrementing any applicable performance counters. C.NOP expands to `nop`. C.NOP is only valid when imm=0; **Special Case**: The code points with imm≠0 encode HINTs. **Exception**: none
RVU.16.123	16.5 (p.110)	H	Breakpoint Instruction
RVU.16.124	16.5 (p.110) 16.8 (p.113) Table 16.7	R	C.EBREAK Instructions Environment break instruction **Encoding**: CR-Type  **Valid Base**: RV32, RV64, RV128 **Task**: – **Explanation**: 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. **Special Case**: none **Exception**: none
RVU.16.125	16.6 (p.110)	H	Usage of C Instructions in LR/SC Sequences
RVU.16.126	16.6 (p.110)	R	On implementations that support the C extension, compressed forms of the I instructions permitted inside constrained LR/SC sequences, as described in Section 8.3, are also permitted inside constrained LR/SC sequences.
RVU.16.127	16.6 (p.110)	C	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.
RVU.16.128	16.7 (p.110)	H	HINT Instructions
RVU.16.129	16.7 (p.110)	R	A portion of the RVC encoding space is reserved for microarchitectural HINTs. Like the HINTs in the RV32I base ISA (see Section 2.9), these instructions do not modify any architectural state, except for advancing the pc and any applicable performance counters. HINTs are executed as no-ops on implementations that ignore them.
RVU.16.130	16.7 (p.110)	I	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).

ID	REFERENCE	TYPE	DEFINITION
RVU.16.131	16.7 (p.111)	C	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.
RVU.16.132	16.7 (p.111)	R	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.
RVU.16.133	16.7 (p.111)	C	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.
RVU.16.134	16.7 (p.111) Table 16.3	R	Following HINT instructions are reserved for standard HINTs. but none are presently defined.

Instruction	Constraints	Code Points
C.NOP	imm≠0	63
C.ADDI	rd≠x0 and imm=0	31
C.LI	rd=x0	64
C.LUI	rd=x0 and imm≠0	63
C.MV	rd=x0 and rs2≠x0	31
C.ADD	rd=x0 and rs2≠x0	31

ID	REFERENCE	TYPE	DEFINITION
RVU.16.135	16.7 (p.111) Table 16.3	R	Following HINT instructions are reserved for custom HINTs: no standard HINTs will ever be defined in this subspace.

Instruction	Constraints	Code Points
C.SLLI	rd=x0, imm≠0	31 (RV32), 63 (RV64, RV128)
C.SLLI64	rd=x0	1
C.SLLI64	rd≠x0, (RV32, RV64)	31 (RV32, RV64), 0 (RV128)
C.SRLI64	(RV32, RV64)	8 (RV32, RV64), 0 (RV128)
C.SRAI64	(RV32, RV64)	8 (RV32, RV64), 0 (RV128)

ID	REFERENCE	TYPE	DEFINITION
RVU.16.136	16.8 (p.112)	H	RVC Instruction Set Listings

ID	REFERENCE	TYPE	DEFINITION
RVU.16.137	16.8 (p.112) Table 16.4	I	Table below shows a map of the major opcodes for RVC. Each row of the table corresponds to one quadrant of the encoding space. The last quadrant, which has the two least-significant bits set, corresponds to instructions wider than 16 bits, including those in the base ISAs.

inst[15:13] / inst[1:0]	000	001	010	011	100	101	110	111	
00	ADDI4 SPN	FLD	LW	FLW	Reserved	FSD	SW	FSW	RV32
		FLD		LD		FSD		SD	RV64
		LQ		LD		SQ		SD	RV128
01	ADDI	JAL	LI	LUI/ ADDI16 SP	MISC-ALU	J	BEQZ	BNEZ	RV32
		ADDIW							RV64
		ADDIW							RV128
10	SLLI	FLDSP	LWSP	FLWSP	J[AL]R/ MV/ ADD	FSDSP	SWSP	FSWSP	RV32
		FLDSP		LDSP		FSDSP		SDSP	RV64
		LQSP		LDSP		SQSP		SDSP	RV128
11	>16b								

RVU.16.138	16.8 (p.112)	I	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 custom extensions; or HINT to indicate that the opcode is reserved for microarchitectural hints

CHAPTER 22 "Zam" Standard Extension for Misaligned Atomics

ID	REFERENCE	TYPE	DEFINITION
RVU.22.1	22.0 (p.125)	H	"Zam" Standard Extension for Misaligned Atomics
RVU.22.2	22.0 (p.125) preface (p.i)	I	Zam extension version is 0.1 and status is draft.
RVU.22.3	22.0 (p.125)	I	This chapter defines the "Zam" extension, which extends the "A" extension by standardizing support for misaligned atomic memory operations (AMOs).
RVU.22.4	22.0 (p.125)	R	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.
RVU.22.5	22.0 (p.125)	R	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.
RVU.22.6	22.0 (p.125)	I	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.
RVU.22.7	22.0 (p.125)	R	Aligned instructions under "Zam" continue to behave as they normally do under RVWMO
RVU.22.8	22.0 (p.125)	C	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 dependencies) must behave as if the AMO is still a single memory operation.

CHAPTER 23 “Ztso” Standard Extension for Total Store Ordering

ID	REFERENCE	TYPE	DEFINITION
RVU.23.1	23.0 (p.127)	H	“Ztso” Standard Extension for Total Store Ordering
RVU.23.2	23.0 (p.127) preface (p.i)	I	Ztso extension version is 0.1 and status is frozen.
RVU.23.3	23.0 (p.127)	I	This chapter defines the “Ztso” extension for the RISC-V Total Store Ordering (RVTSO) memory consistency model.
RVU.23.4	23.0 (p.127)	R	RVTSO is defined as a delta from RVWMO, which is defined in Chapter 14.1.
RVU.23.5	23.0 (p.127)	C	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.
RVU.23.6	23.0 (p.127)	R	Under RVTSO model, all load operations behave as if they have an acquire-RCpc annotation
RVU.23.7	23.0 (p.127)	R	Under RVTSO model, all store operations behave as if they have a release-RCpc annotation.
RVU.23.8	23.0 (p.127)	R	Under RVTSO model, all AMOs behave as if they have both acquire-RCsc and release-RCsc annotations
RVU.23.9	23.0 (p.127)	C	These rules render all PPO rules except 4–7 redundant. They also make redundant any non-I/O fences that do not have both PW and SR set. Finally, they also imply that no memory operation will be reordered past an AMO in either direction.
RVU.23.10	23.0 (p.127)	C	In the context of RVTSO, as is the case for RVWMO, the storage ordering annotations are concisely and completely defined by PPO rules 5–7. In both of these memory models, it is the Load Value Axiom that allows a hart to forward a value from its store buffer to a subsequent (in program order) load—that is to say that stores can be forwarded locally before they are visible to other harts.
RVU.23.11	23.0 (p.127)	I	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.
RVU.23.12	23.0 (p.127)	R	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 24 RV32/64G Instruction Set Listings

ID	REFERENCE	TYPE	DEFINITION
RVU.24.1	24.0 (p.129)	H	RV32/64G Instruction Set Listings
RVU.24.2	24.0 (p.129)	I	One goal of the RISC-V project is that it be used as a stable software development target. For this purpose, we define a combination of a base ISA (RV32I or RV64I) plus selected standard extensions (IMAFD, Zicsr, Zifencei) as a “general-purpose” ISA, and we use the abbreviation G for the IMAFDZicsr Zifencei combination of instruction-set extensions. This chapter presents opcode maps and instruction-set listings for RV32G and RV64G.
RVU.24.3	24.0 (p.129)	R	RV32G contains RV32I, RV32M, RV32A, RV32F, RV32D, Zicsr and Zifencei extensions
RVU.24.4	24.0 (p.129)	R	RV64G contains RV64I, RV64M, RV64A, RV64F, RV64D, Zicsr and Zifencei extensions
RVU.24.5	24.0 (p.129) Table 24.1	R	RISC-V base opcode map, when inst[1:0]=11

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111
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	≥80b

RVU.24.6	24.0 (p.129)	R	Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits.
RVU.24.7	24.0 (p.129)	R	Opcodes marked as *reserved* should be avoided for custom instruction-set extensions as they might be used by future standard extensions.
RVU.24.8	24.0 (p.129)	R	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.
RVU.24.9	24.0 (p.129)	R	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.
RVU.24.10	24.0 (p.129)	I	We believe RV32G and RV64G provide simple but complete instruction sets for a broad range of general-purpose computing.
RVU.24.11	24.0 (p.129)	R	The optional compressed instruction set described in Chapter 16 can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

ID	REFERENCE	TYPE	DEFINITION
RVU.24.12	24.0 (p.129)	I	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 26 has a more extensive discussion of ways to add extensions to the RISC-V ISA.