The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1

Andrew Waterman
Yunsup Lee
David A. Patterson
Krste Asanović

Electrical Engineering and Computer Sciences
University of California at Berkeley

May 31, 2016

The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Version 2.1

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu
May 31, 2016

Preface

This is version 2.1 of the document describing the RISC-V user-level architecture. Note the frozen user-level ISA base and extensions IMAFDQ version 2.0 have not changed from the previous version of this document [30], but some specification holes have been fixed and the documentation has been improved. Some changes have been made to the software conventions.

- Numerous additions and improvements to the commentary sections.
- Separate version numbers for each chapter.
- Modification to long instruction encodings >64 bits to avoid moving the *rd* specifier in very long instruction formats.
- CSR instructions are now described in the base integer format where the counter registers are introduced, as opposed to only being introduced later in the floating-point section (and the companion privileged architecture manual).
- The SCALL and SBREAK instructions have been renamed to ECALL and EBREAK, respectively. Their encoding and functionality are unchanged.
- Clarification of floating-point NaN handling, and a new canonical NaN value.
- Clarification of values returned by floating-point to integer conversions that overflow.
- Clarification of LR/SC allowed successes and required failures, including use of compressed instructions in the sequence.
- A new RV32E base ISA proposal for reduced integer register counts.
- A revised calling convention.
- Relaxed stack alignment for soft-float calling convention, and description of the RV32E calling convention.
- A revised proposal for the C compressed extension, version 1.9.

Preface to Version 2.0

This is the second release of the user ISA specification, and we intend the specification of the base user ISA plus general extensions (i.e., IMAFD) to remain fixed for future development. The following changes have been made since Version 1.0 [29] of this ISA specification.

- The ISA has been divided into an integer base with several standard extensions.
- The instruction formats have been rearranged to make immediate encoding more efficient.
- The base ISA has been defined to have a little-endian memory system, with big-endian or bi-endian as non-standard variants.

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

Contents

Chapter 1

Introduction

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

- A completely *open* ISA that is freely available to academia and industry.
- A *real* ISA suitable for direct native hardware implementation, not just simulation or binary translation.
- An ISA that avoids "over-architecting" for a particular microarchitecture style (e.g., microcoded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
- An ISA separated into a *small* base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general-purpose software development.
- Support for the revised 2008 IEEE-754 floating-point standard [10].
- An ISA supporting extensive user-level ISA extensions and specialized variants.
- Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
- An ISA with support for highly-parallel multicore or manycore implementations, including heterogeneous multiprocessors.
- Optional *variable-length instructions* to both expand available instruction encoding space and to support an optional *dense instruction encoding* for improved performance, static code size, and energy efficiency.
- A fully virtualizable ISA to ease hypervisor development.
- An ISA that simplifies experiments with new supervisor-level and hypervisor-level ISA designs.

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

The name RISC-V was chosen to represent the fifth major RISC ISA design from UC Berkeley (RISC-I [18], RISC-II [11], SOAR [27], and SPUR [14] were the first four). We also pun on the

use of the Roman numeral "V" to signify "variations" and "vectors", as support for a range of architecture research, including various data-parallel accelerators, is an explicit goal of the ISA design.

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

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

- **Commercial ISAs are proprietary.** *Except for SPARC V8, which is an open IEEE standard [1], most owners of commercial ISAs carefully guard their intellectual property and do not welcome freely available competitive implementations. This is much less of an issue for academic research and teaching using only software simulators, but has been a major concern for groups wishing to share actual RTL implementations. It is also a major concern for entities who do not want to trust the few sources of commercial ISA implementations, but who are prohibited from creating their own clean room implementations. We cannot guarantee that all RISC-V implementations will be free of third-party patent infringements, but we can guarantee we will not attempt to sue a RISC-V implementor.*

- **Commercial ISAs are only popular in certain market domains.** *The most obvious examples at time of writing are that the ARM architecture is not well supported in the server space, and the Intel x86 architecture (or for that matter, almost every other architecture) is not well supported in the mobile space, though both Intel and ARM are attempting to enter each other's market segments. Another example is ARC and Tensilica, which provide extensible cores but are focused on the embedded space. This market segmentation dilutes the benefit of supporting a particular commercial ISA as in practice the software ecosystem only exists for certain domains, and has to be built for others.*

- **Commercial ISAs come and go.** *Previous research infrastructures have been built around commercial ISAs that are no longer popular (SPARC, MIPS) or even no longer in production (Alpha). These lose the benefit of an active software ecosystem, and the lingering intellectual property issues around the ISA and supporting tools interfere with the ability of interested third parties to continue supporting the ISA. An open ISA might also lose popularity, but any interested party can continue using and developing the ecosystem.*

- **Popular commercial ISAs are complex.** *The dominant commercial ISAs (x86 and ARM) are both very complex to implement in hardware to the level of supporting common software stacks and operating systems. Worse, nearly all the complexity is due to bad, or at least outdated, ISA design decisions rather than features that truly improve efficiency.*

- **Commercial ISAs alone are not enough to bring up applications.** *Even if we expend the effort to implement a commercial ISA, this is not enough to run existing applications for that ISA. Most applications need a complete ABI (application binary interface) to run, not just the user-level ISA. Most ABIs rely on libraries, which in turn rely on operating system support. To run an existing operating system requires implementing the supervisor-level ISA and device interfaces expected by the OS. These are usually much less well-specified and considerably more complex to implement than the user-level ISA.*

- **Popular commercial ISAs were not designed for extensibility.** *The dominant commercial ISAs were not particularly designed for extensibility, and as a consequence have added considerable instruction encoding complexity as their instruction sets have grown. Companies such as Tensilica (acquired by Cadence) and ARC (acquired by Synopsys) have built ISAs and toolchains around extensibility, but have focused on embedded applications rather than general-purpose computing systems.*

- **A modified commercial ISA is a new ISA.** *One of our main goals is to support architecture research, including major ISA extensions. Even small extensions diminish the benefit of using a standard ISA, as compilers have to be modified and applications rebuilt from source code to use the extension. Larger extensions that introduce new architectural state also require modifications to the operating system. Ultimately, the modified commercial ISA becomes a new ISA, but carries along all the legacy baggage of the base ISA.*

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

We are far from the first to contemplate an open ISA design suitable for hardware implementation. We also considered other existing open ISA designs, of which the closest to our goals was the OpenRISC architecture [17]. We decided against adopting the OpenRISC ISA for several technical reasons:

- *OpenRISC has condition codes and branch delay slots, which complicate higher performance implementations.*

- *OpenRISC uses a fixed 32-bit encoding and 16-bit immediates, which precludes a denser instruction encoding and limits space for later expansion of the ISA.*

- *OpenRISC does not support the 2008 revision to the IEEE 754 floating-point standard.*

- *The OpenRISC 64-bit design had not been completed when we began.*

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

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

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

1.1   RISC-V ISA Overview

The RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISA is very similar to that of the early

RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. The base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional supervisor-level operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are two primary base integer variants, RV32I and RV64I, described in Chapters 2 and 4, which provide 32-bit or 64-bit user-level address spaces respectively. Hardware implementations and operating systems might provide only one or both of RV32I and RV64I for user programs. Chapter 3 describes the RV32E subset variant of the RV32I base instruction set, which has been added to support small microcontrollers. Chapter 19 describes a future RV128I variant of the base integer instruction set supporting a flat 128-bit user address space.

> Although 64-bit address spaces are a requirement for larger systems, we believe 32-bit address spaces will remain adequate for many embedded and client devices for decades to come and will be desirable to lower memory traffic and energy consumption. In addition, 32-bit address spaces are sufficient for educational purposes. A larger flat 128-bit address space might eventually be required, so we ensured this could be accommodated within the RISC-V ISA framework.

The base integer ISA may be subset by a hardware implementation, but opcode traps and software emulation by a more privileged layer must then be used to implement functionality not provided by hardware.

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

RISC-V has been designed to support extensive customization and specialization. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined. We divide RISC-V instruction-set extensions into *standard* and *non-standard* extensions. Standard extensions should be generally useful and should not conflict with other standard extensions. Non-standard extensions may be highly specialized, or may conflict with other standard or non-standard extensions. Instruction-set extensions may provide slightly different functionality depending on the width of the base integer instruction set. Chapter 10 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 11.

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

standard double-precision floating-point extension, denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores. An integer base plus these four standard extensions ("IMAFD") is given the abbreviation "G" and provides a general-purpose scalar instruction set. RV32G and RV64G are currently the default target of our compiler toolchains. Later chapters describe these and other planned standard RISC-V extensions.

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

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

1.2   Instruction Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction *parcels* in length and parcels are naturally aligned on 16-bit boundaries. The standard compressed ISA extension described in Chapter 14 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.

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

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

An implementation of the standard G ISA need only hold the most-significant 30 bits in instruction caches (a 6.25% saving). On instruction cache refills, any instructions encountered with either low bit clear should be recoded into illegal 30-bit instructions before storing in the cache to preserve illegal instruction exception behavior.

*Perhaps more importantly, by condensing our base ISA into a subset of the 32-bit instruc-
tion word, we leave more space available for custom extensions. In particular, the base RV32I
ISA uses less than 1/8 of the encoding space in the 32-bit instruction word. As described in
Chapter 10, an implementation that does not require support for the standard compressed in-
struction extension can map 3 additional 30-bit instruction spaces into the 32-bit fixed-width
format, while preserving support for standard >=32-bit instruction-set extensions. Further, if
the implementation also does not need instructions >32-bits in length, it can recover a further
four major opcodes.*

---

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

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

---

*We chose little-endian byte ordering for the RISC-V memory system because little-endian sys-
tems 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 nat-
ural for hardware designers. However, certain application areas, such as IP networking, operate
on big-endian data structures, and so we leave open the possibility of non-standard big-endian
or bi-endian systems.*

*We have to fix the order in which instruction parcels are stored in memory, independent
of memory system endianness, to ensure that the length-encoding bits always appear first in
halfword address order. This allows the length of a variable-length instruction to be quickly
determined by an instruction fetch unit by examining only the first few bits of the first 16-bit
instruction parcel. Once we had decided to fix on a little-endian memory system and instruction
parcel ordering, this naturally led to placing the length-encoding bits in the LSB positions of the
instruction format to avoid breaking up opcode fields.*

1.3   Exceptions, Traps, and Interrupts

We use the term *exception* to refer to an unusual condition occurring at run time associated with
an instruction in the current RISC-V thread. We use the term *trap* to refer to the synchronous
transfer of control to a trap handler caused by an exceptional condition occurring within a RISC-V
thread. Trap handlers usually execute in a more privileged environment.

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

The instruction descriptions in following chapters describe conditions that raise an exception dur-
ing execution. Whether and how these are converted into traps is dependent on the execution

environment, though the expectation is that most environments will take a *precise* trap when an exception is signaled (except for floating-point exceptions, which, in the standard floating-point extensions, do not cause traps).

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

		`xxxxxxxxxxxxxxaa`	16-bit (`aa` $\neq$ 11)
	`xxxxxxxxxxxxxxxx`	`xxxxxxxxxxxbbb11`	32-bit (`bbb` $\neq$ 111)
$\cdots$`xxxx`	`xxxxxxxxxxxxxxxx`	`xxxxxxxxxx011111`	48-bit
$\cdots$`xxxx`	`xxxxxxxxxxxxxxxx`	`xxxxxxxxx0111111`	64-bit
$\cdots$`xxxx`	`xxxxxxxxxxxxxxxx`	`xnnnxxxxx1111111`	(80+16*`nnn`)-bit, `nnn`$\neq$111
$\cdots$`xxxx`	`xxxxxxxxxxxxxxxx`	`x111xxxxx1111111`	Reserved for $\geq$192-bits

Byte Address: base+4   base+2   base
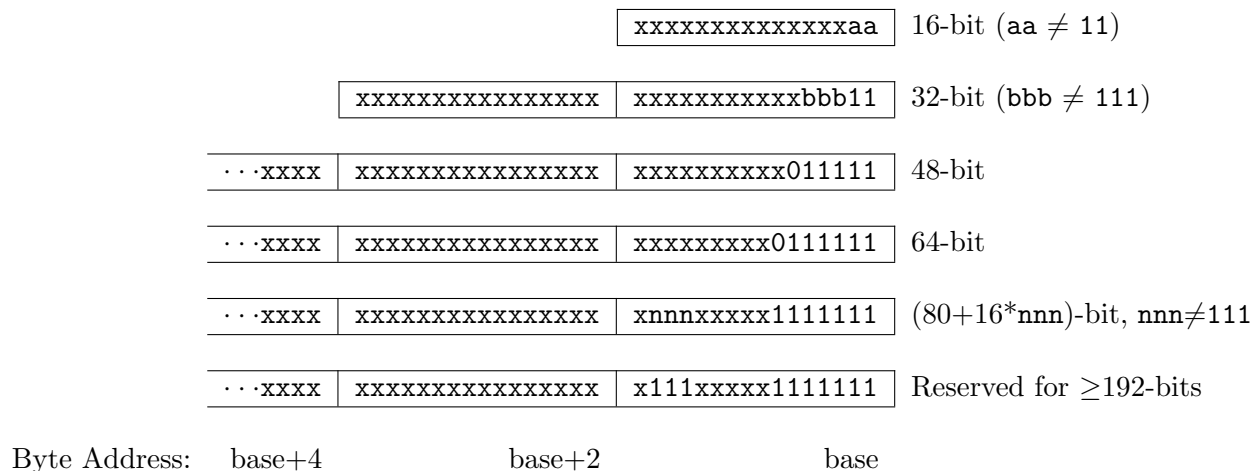
Figure 1.1: RISC-V instruction length encoding.

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

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

Chapter 2

RV32I Base Integer Instruction Set, Version 2.0

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

> *RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 47 unique instructions, though a simple implementation might cover the eight SCALL/SBREAK/CSRR* instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE and FENCE.I instructions as NOPs, reducing hardware instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).*

2.1 Programmers' Model for Base Integer Subset

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

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

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

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

For these reasons, we chose a conventional size of 32 integer registers for the base ISA. Dynamic register usage tends to be dominated by a few frequently accessed registers, and regfile implementations can be optimized to reduce access energy for the frequently accessed registers [26]. The optional compressed 16-bit instruction format mostly only accesses 8 registers and hence can provide a dense instruction encoding, while additional instruction-set extensions could support a much larger register space (either flat or hierarchical) if desired.

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

XLEN-1   0

| x0 / zero |
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

XLEN

XLEN-1   0

| pc |

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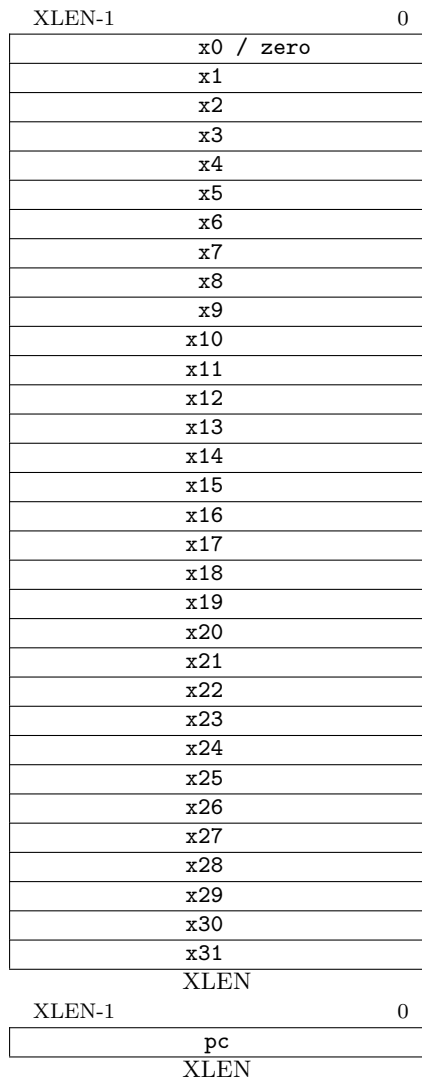
Figure 2.1: RISC-V user-level base integer register state.

2.2 Base Instruction Formats

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

31	25 24	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[31:12]				rd	opcode		U-type
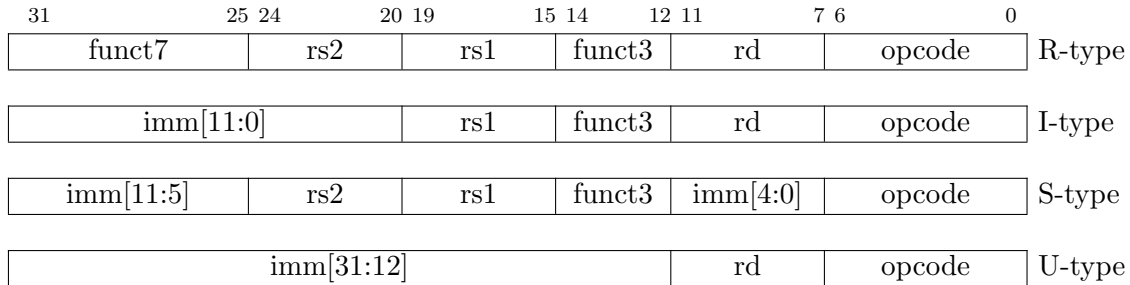
Figure 2.2: RISC-V base instruction formats.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Immediates are packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR [14]).

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. In addition, these immediates are all sign-extended. We did not observe a benefit to using zero-extension for some immediates and wanted to keep the ISA as simple as possible.

2.3 Immediate Encoding Variants

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

In Figure 2.3 each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done. Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

The only difference between the S and SB formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the SB 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])

31	30	25 24	21 20	19	15 14	12 11	8 7	6	0	
funct7		rs2		rs1	funct3	rd		opcode		R-type
imm[11:0]				rs1	funct3	rd		opcode		I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode		S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode		SB-type
imm[31:12]						rd		opcode		U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode		UJ-type

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

31	30	20 19	12 11	10	5 4	1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]		I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]		S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0		B-immediate
inst[31]	inst[30:20]	inst[19:12]		— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0		J-immediate

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

and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in SB format.

Similarly, the only difference between the U and UJ 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 UJ format immediates is chosen to maximize overlap with the other formats and with each other.

Sign-extension is one of the most critical operations on immediates (particularly in RV64I), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

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

> *small additional overhead, but the most common short forward branches have straightforward immediate encodings.*

2.4 Integer Computational Instructions

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

We did not include special instruction set support for overflow checks on integer arithmetic operations, 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. Similarly, signed array bounds checking requires only a single branch instruction. Overflow checks for signed addition require several instructions depending on whether the addend is an immediate or a variable. We considered adding branches that test if the sum of their signed register operands would overflow, but ultimately chose to omit these from the base ISA.

Integer Register-Immediate Instructions

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

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

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

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

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

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

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC
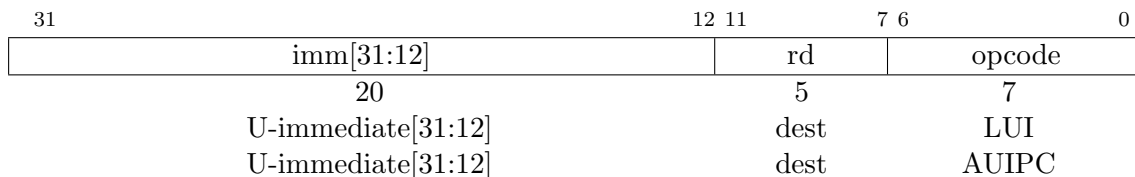
LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register *rd*, filling in the lowest 12 bits with zeros.

AUIPC (add upper immediate to `pc`) is used to build `pc`-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the `pc`, then places the result in register *rd*.

The AUIPC instruction supports two-instruction sequences to access arbitrary offsets from the PC for both control-flow transfers and data accesses. The combination of an AUIPC and the 12-bit immediate in a JALR can transfer control to any 32-bit PC-relative address, while an AUIPC plus the 12-bit immediate offset in regular load or store instructions can access any 32-bit PC-relative data address.

The current PC can be obtained by setting the U-immediate to 0. Although a JAL +4 instruction could also be used to obtain the PC, it might cause pipeline breaks in simpler microarchitectures or pollute the BTB structures in more complex microarchitectures.

Integer Register-Register Operations

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

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

ADD and SUB perform addition and subtraction respectively. Overflows are ignored and the low XLEN bits of results are written to the destination. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if $rs1 < rs2$, 0 otherwise. Note, SLTU *rd, x0, rs2* sets *rd* to 1

if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudo-op SNEZ *rd, rs*). AND, OR, and XOR perform bitwise logical operations.

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

NOP Instruction

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

The NOP instruction does not change any user-visible state, except for advancing the `pc`. NOP is encoded as ADDI *x0, x0, 0*.

NOPs can be used to align code segments to microarchitecturally significant address boundaries, or to leave space for inline code modifications. Although there are many possible ways to encode a NOP, we define a canonical NOP encoding to allow microarchitectural optimizations as well as for more readable disassembly output.

2.5 Control Transfer Instructions

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

Unconditional Jumps

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

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

31	30	21	20	19	12 11	7 6	0
imm[20]	imm[10:1]		imm[11]	imm[19:12]	rd	opcode	
1	10		1	8		5	7
	offset[20:1]					dest	JAL

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

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

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

> *The unconditional jump instructions all use PC-relative addressing to help support position-independent code. The JALR instruction was defined to enable a two-instruction sequence to jump anywhere in a 32-bit absolute address range. A LUI instruction can first load* rs1 *with the upper 20 bits of a target address, then JALR can add in the lower bits. Similarly, AUIPC then JALR can jump anywhere in a 32-bit* pc*-relative address range.*
>
> *Note that the JALR instruction does not treat the 12-bit immediate as multiples of 2 bytes, unlike the conditional branch instructions. This avoids one more immediate format in hardware. In practice, most uses of JALR will have either a zero immediate or be paired with a LUI or AUIPC, so the slight reduction in range is not significant.*
>
> *The JALR instruction ignores the lowest bit of the calculated target address. This 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.*
>
> *Instruction fetch misaligned exceptions are not possible on machines that support extensions with 16-bit aligned instructions, such as the compressed instruction set extension, C.*
>
> *Return-address prediction stacks are a common feature of high-performance instruction-fetch units. We note that* rd *and* rs1 *can be used to guide an implementation's instruction-fetch prediction logic, indicating whether JALR instructions should push (*rd=x1*), pop (*rd=x0, rs1=x1*), or not touch (otherwise) a return-address stack. Similarly, a JAL instruction should push the return address onto the return-address stack only when* rd=x1*.*
>
> *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.*

Conditional Branches

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

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

Branch instructions compare two registers. BEQ and BNE take the branch if registers *rs1* and *rs2* are equal or unequal respectively. BLT and BLTU take the branch if *rs1* is less than *rs2*, using

signed and unsigned comparison respectively. BGE and BGEU take the branch if *rs1* is greater than or equal to *rs2*, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

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

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

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

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

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

We note that various microarchitectural techniques exist to dynamically convert unpredictable short forward branches into internally predicated code to avoid the cost of flushing pipelines on a branch mispredict [9, 13, 12] and have been implemented in commercial processors [22]. 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 [22].

2.6   Load and Store Instructions

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

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

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	
7	5	5	3	5	7	
offset[11:5]	src	base	width	offset[4:0]	STORE	
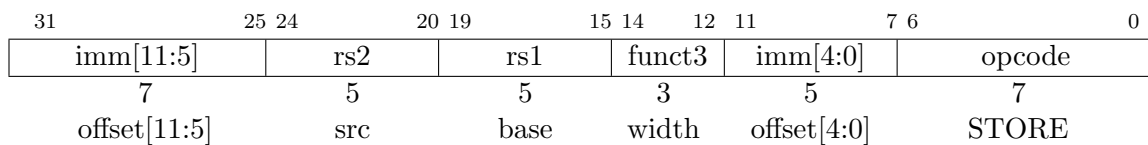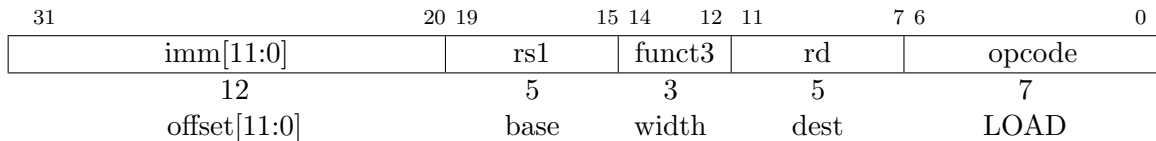
Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register *rs1* to the sign-extended 12-bit offset. Loads copy a value from memory to register *rd*. Stores copy the value in register *rs2* to memory.

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

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

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

take considerable space in the instruction encoding or require very simplified addressing modes (e.g., register indirect only).

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

2.7 Memory Model

The base RISC-V ISA supports multiple concurrent threads of execution within a single user address space. Each RISC-V thread has its own user register state and program counter, and executes an independent sequential instruction stream. The execution environment will define how RISC-V threads are created and managed. RISC-V threads can communicate and synchronize with other threads either via calls to the execution environment, which are documented separately in the specification for each execution environment, or directly via the shared memory system. RISC-V threads can also interact with I/O devices, and indirectly with each other, via loads and stores to portions of the address space assigned to I/O.

In the base RISC-V ISA, each RISC-V thread observes its own memory operations as if they executed sequentially in program order. RISC-V has a relaxed memory model between threads, requiring an explicit FENCE instruction to guarantee any specific ordering between memory operations from different RISC-V threads. Chapter 6 describes the optional atomic memory instruction extensions "A", which provide additional synchronization operations.

31	28	27	26	25	24	23	22	21	20	19	15	14	12	11	7	6	0
0		PI	PO	PR	PW	SI	SO	SR	SW	rs1		funct3		rd		opcode	
4		1	1	1	1	1	1	1	1	5		3		5		7	
0		predecessor				successor				0		FENCE		0		MISC-MEM	

The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V threads and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V thread or external device can observe any operation in the *successor* set following a FENCE before any operation in the *predecessor* set preceding the FENCE. The execution environment will define what I/O operations are possible, and in particular, which load and store instructions might be treated and ordered as device input and device output operations respectively rather than memory reads and writes. For example, memory-mapped I/O devices will typically be accessed with uncached loads and stores that are ordered using the I and O bits rather than the R and W bits. Instruction-set extensions might also describe new coprocessor I/O instructions that will also be ordered using the I and O bits in a FENCE.

The unused fields in the FENCE instruction, *imm[11:8]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

> *We chose a relaxed memory model to allow high performance from simple machine implementations. A relaxed memory model is also most compatible with 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 thread and also to support alternative non-memory paths to control added coprocessors or I/O devices. Simple implementations may additionally ignore the* predecessor *and* successor *fields and always execute a conservative fence on all operations.*

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

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on the same RISC-V thread until a FENCE.I instruction is executed. A FENCE.I instruction only ensures that a subsequent instruction fetch on a RISC-V thread will see any previous data stores already visible to the same RISC-V thread. FENCE.I does *not* ensure that other RISC-V threads' instruction fetches will observe the local thread's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V threads, the writing thread has to execute a data FENCE before requesting that all remote RISC-V threads execute a FENCE.I.

The unused fields in the FENCE.I instruction, *imm[11:0]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

> *The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, then only the pipeline needs to be flushed at a FENCE.I.*
>
> *We considered but did not include a "store instruction word" instruction (as in MAJC [25]). JIT compilers may generate a large trace of instructions before a single FENCE.I, and amortize any instruction cache snooping/invalidation overhead by writing translated instructions to memory regions that are known not to reside in the I-cache.*

2.8   Control and Status Register Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions. CSR instructions are described in this section, with the two other user-level SYSTEM instructions described in the following section.

> *The SYSTEM instructions are defined to allow simpler implementations to always trap to a single software trap handler. More sophisticated implementations might execute more of each system instruction in hardware.*

CSR Instructions

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

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

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

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

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

For both CSRRS and CSRRC, if *rs1*=x0, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, such as raising illegal instruction exceptions on accesses to read-only CSRs. Note that if *rs1* specifies a register holding a zero value other than x0, the instruction will still attempt to write the unmodified value back to the CSR and will cause any attendant side effects.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit immediate (zimm[4:0]) field encoded in the *rs1* field instead of a value from an integer register. For CSRRSI and CSRRCI, if the zimm[4:0] field is zero, then these instructions will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write. For CSRRWI, if *rd*=x0, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

Some CSRs, such as the instructions retired counter, `instret`, may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes a CSR, the update occurs after the execution of the instruction. In particular, a value written to `instret` by one instruction will be the value read by the following instruction (i.e., the increment of `instret` caused by the first instruction retiring happens before the write of the new value).

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

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

Timers and Counters

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

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

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

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

The RDINSTRET pseudo-instruction reads the low XLEN bits of the `instret` CSR, which counts the number of instructions retired by this hardware thread from some arbitrary start point in the past. RDINSTRETH is an RV32I-only instruction that reads bits 63–32 of the same instruction counter. The underlying 64-bit counter that should never overflow in practice.

The following code sequence will read a valid 64-bit cycle counter value into `x3:x2`, even if the counter overflows between reading its upper and lower halves.

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

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

We mandate these basic counters be provided in all implementations as they are essential for basic performance analysis, adaptive and dynamic optimization, and to allow an application to work with real-time streams. Additional counters should be provided to help diagnose performance problems and these should be made accessible from user-level application code with low overhead.

We required the counters be 64 bits wide, even on RV32, as otherwise it is very difficult for software to determine if values have overflowed. For a low-end implementation, the upper 32 bits of each counter can be implemented using software counters incremented by a trap handler triggered by overflow of the lower 32 bits. The sample code described above shows how the full 64-bit width value can be safely read using the individual 32-bit instructions.

In some applications, it is important to be able to read multiple counters at the same instant in time. When run under a multitasking environment, a user thread can suffer a context switch while attempting to read the counters. One solution is for the user thread to read the real-time counter before and after reading the other counters to determine if a context switch occurred in the middle of the sequence, in which case the reads can be retried. We considered adding output latches to allow a user thread to snapshot the counter values atomically, but this would increase the size of the user context, especially for implementations with a richer set of counters.

2.9 Environment Call and Breakpoints

funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file.

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

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

Chapter 3

RV32E Base Integer Instruction Set, Version 1.9

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

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

3.1   RV32E Programmers' Model

RV32E reduces the integer register count to 16 general-purpose registers, (`x0`–`x15`), where `x0` is a dedicated zero register.

> *We have found that in the small RV32I core designs, the upper 16 registers consume around one quarter of the total area of the core excluding memories, thus their removal saves around 25% core area with a corresponding core power reduction.*

> *This change requires a different calling convention and ABI. In particular, RV32E is only used with a soft-float calling convention. Systems with hardware floating-point must use an I base.*

3.2   RV32E Instruction Set

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

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

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

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

3.3   RV32E Extensions

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

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

The privileged architecture of an RV32E system can include user mode as well as machine mode, and the Mbare, Mbb, and Mbbid memory management schemes described in Volume II.

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

Chapter 4

RV64I Base Integer Instruction Set, Version 2.0

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

4.1 Register State

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

4.2 Integer Computational Instructions

Additional instruction variants are provided to manipulate 32-bit values in RV64I, indicated by a 'W' suffix to the opcode. These "*W" instructions ignore the upper 32 bits of their inputs and always produce 32-bit signed values, i.e. bits XLEN-1 through 31 are equal. They cause an illegal instruction exception in RV32I.

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

Integer Register-Immediate Instructions

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

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

imm[11:6]	imm[5]	imm[4:0]	rs1	funct3	rd	opcode
6	1	5	5	3	5	7
000000	shamt[5]	shamt[4:0]	src	SLLI	dest	OP-IMM
000000	shamt[5]	shamt[4:0]	src	SRLI	dest	OP-IMM
010000	shamt[5]	shamt[4:0]	src	SRAI	dest	OP-IMM
000000	0	shamt[4:0]	src	SLLIW	dest	OP-IMM-32
000000	0	shamt[4:0]	src	SRLIW	dest	OP-IMM-32
010000	0	shamt[4:0]	src	SRAIW	dest	OP-IMM-32

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

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

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

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

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

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

Integer Register-Register Operations

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

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

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

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

4.3   Load and Store Instructions

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

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

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

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

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

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

4.4 System Instructions

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

Chapter 5

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

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

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

5.1   Multiplication Operations

31		25 24		20 19		15 14		12 11		7 6		0
funct7		rs2		rs1		funct3		rd		opcode		
7		5		5		3		5		7		
MULDIV		multiplier		multiplicand		MUL/MULH[[S]U]		dest		OP		
MULDIV		multiplier		multiplicand		MULW		dest		OP-32		

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

MULW is only valid for RV64, and multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register. MUL can be used to

obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear.

5.2   Division Operations

funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32

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

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

The semantics for division by zero and division overflow are summarized in Table 5.1. The quotient of division by zero has all bits set, i.e. $2^{XLEN} - 1$ for unsigned division or -1 for signed division. The remainder of division by zero equals the dividend. Signed division overflow occurs only when the most-negative integer, -2^{XLEN-1}, is divided by -1. The quotient of signed division overflow is equal to the dividend, and the remainder is zero. Unsigned division overflow cannot occur.

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

Table 5.1: Semantics for division by zero and division overflow.

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

Chapter 6

"A" Standard Extension for Atomic Instructions, Version 2.0

The standard atomic instruction extension is denoted by instruction subset name "A", and contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V threads running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model [6].

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

6.1 Specifying Ordering of Atomic Instructions

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

To provide more efficient support for release consistency [6], each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V threads. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains.

If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the *aq* bit is set, the atomic memory operation is treated as an *acquire* access, i.e., no following memory operations on this RISC-V thread can be observed to take place before the acquire memory operation. If only the *rl* bit is set, the atomic memory operation is treated as

a *release* access, i.e., the release memory operation can not be observed to take place before any earlier memory operations on this RISC-V thread. 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 thread, and can only be observed by any other thread in the same global order of all sequentially consistent atomic memory operations to the same address domain.

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

6.2   Load-Reserved/Store-Conditional Instructions

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

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

> *Both compare-and-swap (CAS) and LR/SC can be used to build lock-free data structures. After extensive discussion, we opted for LR/SC for several reasons: 1) CAS suffers from the ABA problem, which LR/SC avoids because it monitors all accesses to the address rather than only checking for changes in the data value; 2) CAS would also require a new integer instruction format to support three source operands (address, compare value, swap value) as well as a different memory system message format, which would complicate microarchitectures; 3) Furthermore, to avoid the ABA problem, other systems provide a double-wide CAS (DW-CAS) to allow a counter to be tested and incremented along with a data word. This requires reading five registers and writing two in one instruction, and also a new larger memory system message type, further complicating implementations; 4) LR/SC provides a more efficient implementation of many primitives as it only requires one load as opposed to two with CAS (one load before the CAS instruction to obtain a value for speculative computation, then a second load as part of the CAS instruction to check if value is unchanged before updating).*
>
> *The main disadvantage of LR/SC over CAS is livelock, which we avoid with an architected guarantee of eventual forward progress as described below. Another concern is whether the influence of the current x86 architecture, with its DW-CAS, will complicate porting of synchronization libraries and other software that assumes DW-CAS is the basic machine primitive. A possible mitigating factor is the recent addition of transactional memory instructions to x86, which might cause a move away from DW-CAS.*

The failure code with value 1 is reserved to encode an unspecified failure. Other failure codes are reserved at this time, and portable software should only assume the failure code will be non-zero.

LR and SC operate on naturally-aligned 64-bit (RV64 only) or 32-bit words in memory. Misaligned addresses will generate misaligned address exceptions.

> *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.*

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

> *One advantage of CAS is that it guarantees that some thread eventually makes progress, whereas an LR/SC atomic sequence could livelock indefinitely on some systems. To avoid this concern, we added an architectural guarantee of forward progress to LR/SC atomic sequences. The restrictions on LR/SC sequence contents allows an implementation to capture a cache line on the LR and complete the LR/SC sequence by holding off remote cache interventions for a bounded short time. Interrupts and TLB misses might cause the reservation to be lost, but eventually the atomic sequence can complete. We restricted the length of LR/SC sequences to fit within 64 contiguous instruction bytes in the base ISA to avoid undue restrictions on instruction cache and TLB size and associativity. Similarly, we disallowed other loads and stores within the sequences to avoid restrictions on data cache associativity. The restrictions on branches and jumps limits the time that can be spent in the sequence. Floating-point operations and integer multiply/divide were disallowed to simplify the operating system's emulation of these instructions on implementations lacking appropriate hardware support.*

An implementation can reserve an arbitrary subset of the memory space on each LR and multiple LR reservations might be active simultaneously for a single hart. An SC can succeed if no accesses from other harts to the address can be observed to have occurred between the SC and the last LR in this hart to reserve the address. Note this LR might have had a different address argument, but reserved the SC's address as part of the memory subset. Following this model, in systems with memory translation, an SC is allowed to succeed if the earlier LR reserved the same location using an alias with a different virtual address, but is also allowed to fail if the virtual address is different. The SC must fail if there is an observable memory access from another hart to the address, or if there is an intervening context switch on this hart, or if in the meantime the hart executed a privileged exception-return instruction.

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

LR/SC can be used to construct lock-free data structures. An example using LR/SC to implement a compare-and-swap function is shown in Figure 6.1. If inlined, compare-and-swap functionality need only take three instructions.

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

Figure 6.1: Sample code for compare-and-swap function using LR/SC.

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

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

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

6.3   Atomic Memory Operations

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

The atomic memory operation (AMO) instructions perform read-modify-write operations for multiprocessor synchronization and are encoded with an R-type instruction format. These AMO in-

structions atomically load a data value from the address in *rs1*, place the value into register *rd*, apply a binary operator to the loaded value and the original value in *rs2*, then store the result back to the address in *rs1*. AMOs can either operate on 64-bit (RV64 only) or 32-bit words in memory. For RV64, 32-bit AMOs always sign-extend the value placed in *rd*. The address held in *rs1* must be naturally aligned to the size of the operand (i.e., eight-byte aligned for 64-bit words and four-byte aligned for 32-bit words). If the address is not naturally aligned, a misaligned address exception will be generated.

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

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

To help implement multiprocessor synchronization, the AMOs optionally provide release consistency semantics. If the *aq* bit is set, then no later memory operations in this RISC-V thread can be observed to take place before the AMO. Conversely, if the *rl* bit is set, then other RISC-V threads will not observe the AMO before memory accesses preceding the AMO in this RISC-V thread.

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.*

AMOs can also be used to provide sequentially consistent loads and stores. A sequentially consistent load can be implemented as an AMOADD of x0 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.

An example code sequence for a critical section guarded by a test-and-set spinlock is shown in Figure 6.2. Note the first AMO is marked *aq* to order the lock acquisition before the critical section, and the second AMO is marked *rl* to order the critical section before the lock relinquishment.

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

Figure 6.2: Sample code for mutual exclusion. `a0` contains the address of the lock.

We recommend the use of the AMO Swap idiom shown above for both lock acquire and release to simplify the implementation of speculative lock elision [20].

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

Chapter 7

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

This chapter describes the standard instruction-set extension for single-precision floating-point, which is named "F" and adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard [10].

7.1 F Register State

The F extension adds 32 floating-point registers, `f0`–`f31`, each 32 bits wide, and a floating-point control and status register `fcsr`, which contains the operating mode and exception status of the floating-point unit. This additional state is shown in Figure 7.1. We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, and FLEN=32 for the F single-precision floating-point extension. Most floating-point instructions operate on values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.

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

FLEN-1   0

| f0 |
| f1 |
| f2 |
| f3 |
| f4 |
| f5 |
| f6 |
| f7 |
| f8 |
| f9 |
| f10 |
| f11 |
| f12 |
| f13 |
| f14 |
| f15 |
| f16 |
| f17 |
| f18 |
| f19 |
| f20 |
| f21 |
| f22 |
| f23 |
| f24 |
| f25 |
| f26 |
| f27 |
| f28 |
| f29 |
| f30 |
| f31 |

FLEN

31   0

| fcsr |

32
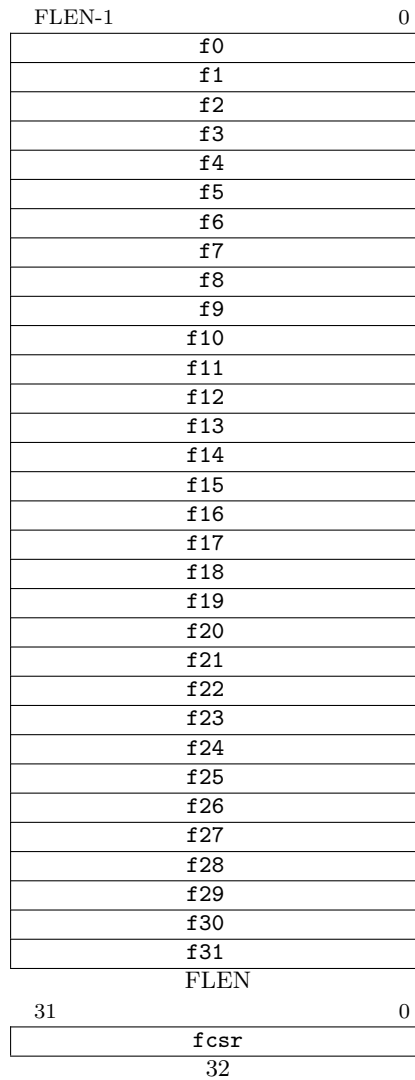
Figure 7.1: RISC-V standard F extension single-precision floating-point state.

31		8 7	5	4	3	2	1	0
0		Rounding Mode (`frm`)		Accrued Exceptions (`fflags`)				
				NV	DZ	OF	UF	NX
24		3		1	1	1	1	1

Figure 7.2: Floating-point control and status register.

7.2 Floating-Point Control and Status Register

The floating-point control and status register, `fcsr`, is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in Figure 7.2.

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

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

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in `frm`. Rounding modes are encoded as shown in Table 7.1. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in `frm`. If `frm` is set to an invalid value (101–111), any subsequent attempt to ex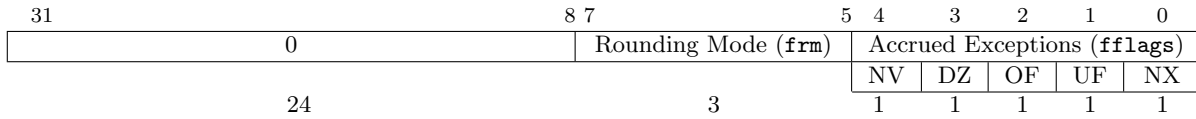ecute a floating-point operation with a dynamic rounding mode will cause an illegal instruction trap. Some instructions that have the *rm* field are nevertheless unaffected by the rounding mode; they should have their *rm* field set to RNE (000).

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

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

Table 7.1: Rounding mode encoding.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 7.2.

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

Table 7.2: Accrued exception flag encoding.

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

7.3 NaN Generation and Propagation

Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN. The canonical NaN has a positive sign and all significand bits clear except the MSB, a.k.a. the quiet bit. For single-precision floating-point, this corresponds to the pattern `0x7fc00000`.

For FMIN and FMAX, if at least one input is a signaling NaN, or if both inputs are quiet NaNs, the result is the canonical NaN. If one operand is a quiet NaN and the other is not a NaN, the result is the non-NaN operand.

The sign-injection instructions (FSGNJ, FSGNJN, FSGNJX) do not canonicalize NaNs; they manipulate the underlying bit patterns directly.

We considered propagating NaN payloads, as is recommended by the standard, but this decision would have increased hardware cost. Moreover, since this feature is optional in the standard, it cannot be used in portable code.

Implementors are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.

7.4   Subnormal Arithmetic

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

In the parlance of the IEEE standard, tininess is detected after rounding—that is, the underflow exception is raised only if the rounded result is subnormal, even if the unrounded result would have been subnormal.

Detecting tininess after rounding results in fewer spurious underflow signals.

7.5   Single-Precision Load and Store Instructions

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

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

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	width	imm[4:0]	opcode	
7	5	5	3	5	7	
offset[11:5]	src	base	W	offset[4:0]	STORE-FP	
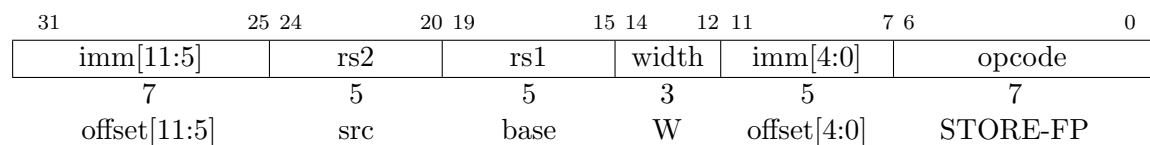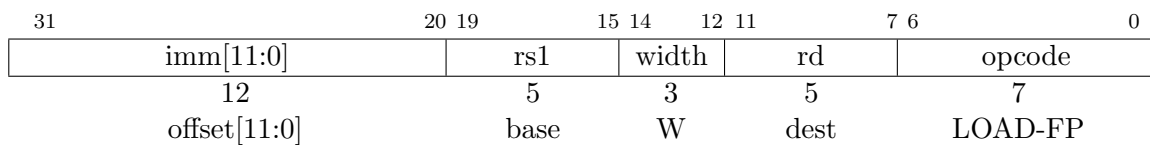
FLW and FSW are only guaranteed to execute atomically if the effective address is naturally aligned.

7.6   Single-Precision Floating-Point Computational Instructions

Floating-point arithmetic instructions with one or two source operands use the R-type format with the OP-FP major opcode. FADD.S, FSUB.S, FMUL.S, and FDIV.S perform single-precision

floating-point addition, subtraction, multiplication, and division, respectively, between *rs1* and *rs2*, writing the result to *rd*. FMIN.S and FMAX.S write, respectively, the smaller or larger of *rs1* and *rs2* to *rd*. FSQRT.S computes the square root of *rs1* and writes the result to *rd*.

The 2-bit floating-point format field *fmt* is encoded as shown in Table 7.3. It is set to S (00) for all instructions in the F extension.

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

Table 7.3: Format field encoding.

All floating-point operations that perform rounding can select the rounding mode using the *rm* field with the encoding shown in Table 7.1.

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

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

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

7.7 Single-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.S or FCVT.L.S converts a floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.S.W

or FCVT.S.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a floating-point number in floating-point register *rd*. FCVT.WU.S, FCVT.LU.S, FCVT.S.WU, and FCVT.S.LU variants convert to or from unsigned integer values. FCVT.L[U].S and FCVT.S.L[U] are illegal in RV32. If the rounded result is not representable in the destination format, it is clipped to the nearest value and the invalid flag is set. Table 7.4 gives the range of valid inputs for FCVT.*int*.S and the behavior for invalid inputs.

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

Table 7.4: Domains of float-to-integer conversions and behavior for invalid inputs.

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

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

Floating-point to floating-point sign-injection instructions, FSGNJ.S, FSGNJN.S, and FSGNJX.S, produce a result that takes all bits except the sign bit from *rs1*. For FSGNJ, the result's sign bit is *rs2*'s sign bit; for FSGNJN, the result's sign bit is the opposite of *rs2*'s sign bit; and for FSGNJX, the sign bit is the XOR of the sign bits of *rs1* and *rs2*. Sign-injection instructions do not set floating-point exception flags. Note, FSGNJ.S *rx, ry, ry* moves *ry* to *rx* (assembler pseudo-op FMV.S *rx, ry*); FSGNJN.S *rx, ry, ry* moves the the negation of *ry* to *rx* (assembler pseudo-op FNEG.S *rx, ry*); and FSGNJX.S *rx, ry, ry* moves the absolute value of *ry* to *rx* (assembler pseudo-op FABS.S *rx, ry*).

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

The sign-injection instructions were proposed by John Hauser. These instructions provide floating-point MV, ABS, and NEG, as well as supporting a few other operations, including the IEEE copySign operation and sign manipulation in transcendental math function libraries. Although MV, ABS, and NEG only need a single register operand, whereas FSGNJ instructions need two, it is unlikely most microarchitectures would add optimizations to benefit from the reduced number of register reads for these relatively infrequent instructions. Even in this

case, a microarchitecture can simply detect when both source registers are the same for FSGNJ instructions and only read a single copy.

Instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.S moves the single-precision value in floating-point register *rs1* represented in IEEE 754-2008 encoding to the lower 32 bits of integer register *rd*. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit. FMV.S.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.

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

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

7.8   Single-Precision Floating-Point Compare Instructions

Floating-point compare instructions perform the specified comparison (equal, less than, or less than or equal) between floating-point registers *rs1* and *rs2* and record the Boolean result in integer register *rd*.

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

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

7.9   Single-Precision Floating-Point Classify Instruction

The FCLASS.S instruction examines the value in floating-point register *rs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in Table 7.5. The corresponding bit in *rd* will be set if the the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in *rd* will be set.

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		fmt		rs2		rs1		rm		rd		opcode	
5		2		5		5		3		5		7	
FCLASS		S		0		src		001		dest		OP-FP	
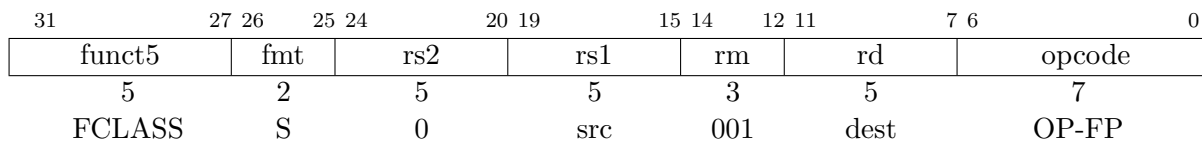
rd bit	Meaning
0	*rs1* is $-\infty$.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is $+0$.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

Table 7.5: Format of result of FCLASS instruction.

Chapter 8

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

This chapter describes the standard double-precision floating-point instruction-set extension, which is named "D" and adds double-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The D extension depends on the base single-precision instruction subset F.

8.1 D Register State

The D extension widens the 32 floating-point registers, `f0`–`f31`, to 64 bits (FLEN=64 in Figure 7.1).

8.2 Double-Precision Load and Store Instructions

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

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

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

49
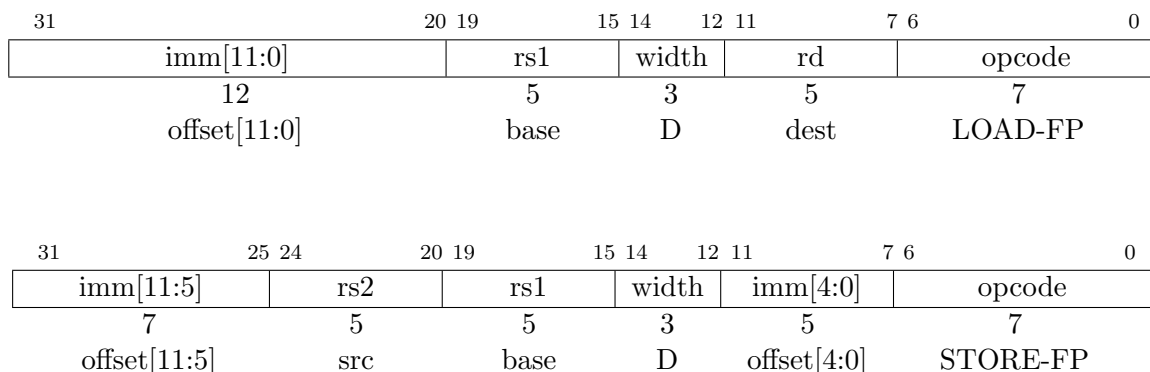
If a floating-point register holds a single-precision value, it is guaranteed that a FSD of that register will place a value into memory that when reloaded with a FLD will recreate the original single-precision value in a register. The data format that is stored in memory is undefined beyond having this property.

User-level code 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. A common case is for callee-save registers, but this is also essential to implement varargs and user-level threading libraries.

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

8.3   Double-Precision Floating-Point Computational Instructions

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

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

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

8.4   Double-Precision Floating-Point Conversion and Move Instructions

Floating-point-to-integer and integer-to-floating-point conversion instructions are encoded in the OP-FP major opcode space. FCVT.W.D or FCVT.L.D converts a double-precision floating-point number in floating-point register *rs1* to a signed 32-bit or 64-bit integer, respectively, in integer register *rd*. FCVT.D.W or FCVT.D.L converts a 32-bit or 64-bit signed integer, respectively, in integer register *rs1* into a double-precision floating-point number in floating-point register *rd*. FCVT.WU.D, FCVT.LU.D, FCVT.D.WU, and FCVT.D.LU variants convert to or from unsigned integer values. FCVT.L[U].D and FCVT.D.L[U] are illegal in RV32. The range of valid inputs for FCVT.*int*.D and the behavior for invalid inputs are the same as for FCVT.*int*.S.

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

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

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

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

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

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

For RV64 only, instructions are provided to move bit patterns between the floating-point and integer registers. FMV.X.D moves the double-precision value in floating-point register *rs1* to a representation in IEEE 754-2008 standard encoding in integer register *rd*. If the last value written to the source floating-point register was a single-precision floating-point value, then the value returned by FMV.X.D is undefined beyond having the property that moving the value back to a floating-point register will recreate the original single-precision value. FMV.D.X moves the double-precision value encoded in IEEE 754-2008 standard encoding from the integer register *rs1* to the floating-point register *rd*.

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

8.5 Double-Precision Floating-Point Compare Instructions

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

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCMP	D	src2	src1	EQ/LT/LE	dest	OP-FP
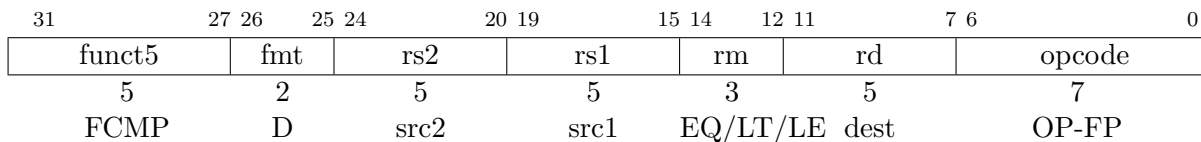
8.6 Double-Precision Floating-Point Classify Instruction

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

funct5	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
FCLASS	D	0	src	001	dest	OP-FP
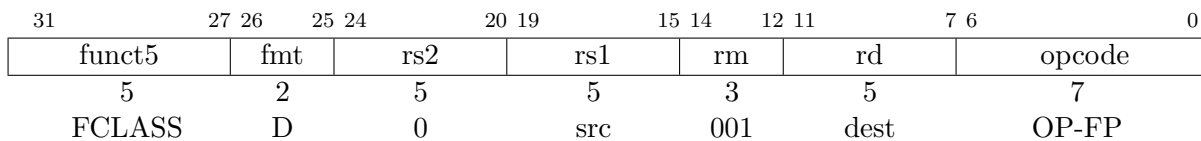
Chapter 9

RV32/64G Instruction Set Listings

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

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

Table 9.1: RISC-V base opcode map, inst[1:0]=11

Table 9.1 shows a map of the major opcodes for RVG. Major opcodes with 3 or more lower bits set are reserved for instruction lengths greater than 32 bits. Opcodes marked as *reserved* should be avoided for custom instruction set extensions as they might be used by future standard extensions. Major opcodes marked as *custom-0* and *custom-1* will be avoided by future standard extensions and are recommended for use by custom instruction-set extensions within the base 32-bit instruction format. The opcodes marked *custom-2/rv128* and *custom-3/rv128* are reserved for future use by RV128, but will otherwise be avoided for standard extensions and so can also be used for custom instruction-set extensions in RV32 and RV64.

We believe RV32G and RV64G provide simple but complete instruction sets for a broad range of general-purpose computing. The optional compressed instruction set described in Chapter 14 can be added (forming RV32GC and RV64GC) to improve performance, code size, and energy efficiency, though with some additional hardware complexity.

As we move beyond IMAFDC into further instruction set extensions, the added instructions tend to be more domain-specific and only provide benefits to a restricted class of applications, e.g., for multimedia or security. Unlike most commercial ISAs, the RISC-V ISA design clearly separates the base ISA and broadly applicable standard extensions from these more specialized additions. Chapter 10 has a more extensive discussion of ways to add extensions to the RISC-V ISA.

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

RV32I Base Instruction Set

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

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

RV64I Base Instruction Set (in addition to RV32I)

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

RV32M Standard Extension

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

RV64M Standard Extension (in addition to RV32M)

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

RV32A Standard Extension

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

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

RV64A Standard Extension (in addition to RV32A)

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

RV32F Standard Extension

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

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

RV64F Standard Extension (in addition to RV32F)

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

RV32D Standard Extension

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

RV64D Standard Extension (in addition to RV32D)

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

Table 9.2: Instruction listing for RISC-V

Chapter 10

Extending RISC-V

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

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

10.1 Extension Terminology

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

Standard versus Non-Standard Extension

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

- A standard extension is one that is generally useful and that is designed to not conflict with any other standard extension. Currently, "MAFDQLCBTPV", described in other chapters of this manual, are either complete or planned standard extensions.

- A non-standard extension may be highly specialized and may conflict with other standard or non-standard extensions. We anticipate a wide variety of non-standard extensions will be developed over time, with some eventually being promoted to standard extensions.

Instruction Encoding Spaces and Prefixes

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

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

Although an instruction encoding space could be of any size, adopting a smaller set of common sizes simplifies packing independently developed extensions into a single global encoding. Table 10.1 gives the suggested sizes for RISC-V.

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

Table 10.1: Suggested standard RISC-V instruction encoding space sizes.

Greenfield versus Brownfield Extensions

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

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

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

Table 10.2: Two-dimensional characterization of standard instruction-set extensions.

Table 10.2 shows the bases and standard extensions placed in a simple two-dimensional taxonomy. One axis is whether the extension is greenfield or brownfield, while the other axis is whether the extension adds architectural state. For greenfield extensions, the size of the instruction encoding space is given in parentheses. For brownfield extensions, the name of the extension (greenfield or brownfield) it builds upon is given in parentheses. Additional user-level architectural state usually implies changes to the supervisor-level system or possibly to the standard calling convention.

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

Standard-Compatible Global Encodings

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

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

Guaranteed Non-Standard Encoding Space

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

10.2   RISC-V Extension Design Philosophy

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

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

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

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

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

10.3   Extensions within fixed-width 32-bit instruction format

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

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

Available 30-bit instruction encoding spaces

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

Available 25-bit instruction encoding spaces

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

There are four major opcodes expressly reserved for custom extensions (Table 9.1), each of which represents a 25-bit encoding space. Two of these are reserved for eventual use in the RV128 base encoding (will be OP-IMM-64 and OP-64), but can be used for standard or non-standard extensions for RV32 and RV64.

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

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

If an implementation does not require instructions longer than 32-bits, then an additional four major opcodes are available (those marked in gray in Table 9.1).

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

Available 22-bit instruction encoding spaces

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

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

Other spaces

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

10.4   Adding aligned 64-bit instruction extensions

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

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

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

10.5   Supporting VLIW encodings

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

Fixed-size instruction group

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

Encoded-Length Groups

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

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

Fixed-Size Instruction Bundles

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

End-of-Group bits in Prefix

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

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

Chapter 11

ISA Subset Naming Conventions

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

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

11.1 Case Sensitivity

The ISA naming strings are case insensitive.

11.2 Base Integer ISA

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

11.3 Instruction Extensions Names

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

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

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

11.4   Version Numbers

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

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

11.5   Non-Standard Extension Names

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

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

11.6   Supervisor-level Instruction Subsets

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

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

11.7   Supervisor-level Extensions

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

11.8 Subset Naming Convention

Table 11.1 summarizes the standardized subset names.

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

Table 11.1: Standard ISA subset names. The table also defines the canonical order in which subset names must appear in the name string, with top-to-bottom in table indicating first-to-last in the name string, e.g., RV32IMAFDQC is legal, whereas RV32IMAFDCQ is not.
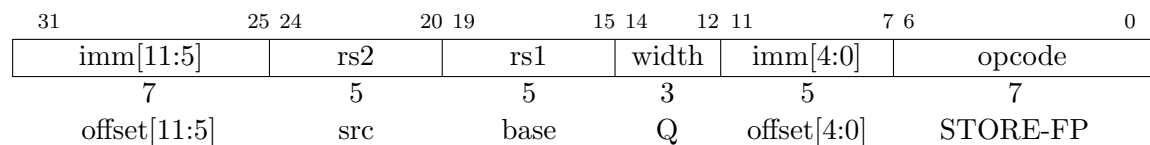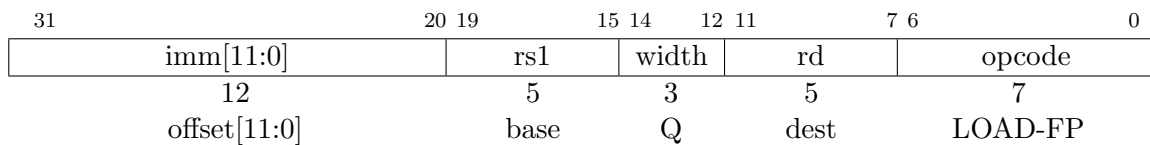
Chapter 12

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

This chapter describes the Q standard extension for 128-bit binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The 128-bit or quad-precision binary floating-point instruction subset is named "Q", and requires RV64IFD. The floating-point registers are now extended to hold either a single, double, or quad-precision floating-point value (FLEN=128).

12.1 Quad-Precision Load and Store Instructions

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

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

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

If a floating-point register holds a single-precision or double-precision value, it is guaranteed that a FSQ of that register will place a value into memory that when reloaded with a FLQ will recreate the original value in a register. The data format that is stored in memory is undefined beyond having this property.

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

12.2   Quad-Precision Computational Instructions

A new supported format is added to the format field of most instructions, as shown in Table 12.1.

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

Table 12.1: Format field encoding.

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

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

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

12.3   Quad-Precision Convert and Move Instructions

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

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

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

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

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

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

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

12.4   Quad-Precision Floating-Point Compare Instructions

Floating-point compare instructions perform the specified comparison (equal, less than, or less than or equal) between floating-point registers *rs1* and *rs2* and record the Boolean result in integer register *rd*.

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

12.5   Quad-Precision Floating-Point Classify Instruction

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

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

Chapter 13

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

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

13.1 Decimal Floating-Point Registers

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

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

Chapter 14

"C" Standard Extension for Compressed Instructions, Version 1.9

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

We believe this draft represents the close to final design for RV32C and RV64C (it seems premature to freeze R128C), though we are requesting one more round of comments, hence the 1.9 revision number. Please send your comments to the `isa-dev` mailing list at `isa-dev@lists.riscv.org`.

14.1 Overview

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

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

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary.

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

The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 14.3, a few opcodes are used for different purposes depending on base ISA width. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.

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

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

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

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

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

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

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

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

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

79

IBM Stretch [3], 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 [2] supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 [23], a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

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

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

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

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and *to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size [28].*

14.2   Compressed Instruction Formats

Table 14.1 shows the eight compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, and CB are limited to just 8 of them. Table 14.2 lists these popular registers, which correspond to registers `x8` to `x15`. Note that there is a separate version of load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow

access to all 32 data registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.

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

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

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

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

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

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

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

Table 14.1: Compressed 16-bit RVC instruction formats.

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

Table 14.2: Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, and CB formats.

14.3 Load and Store Instructions

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

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

Stack-Pointer-Based Loads and Stores

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

These instructions use the CI format.

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

C.LDSP is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `ld rd, offset[8:3](x2)`.

C.LQSP is an RV128C-only instruction that loads a 128-bit value from memory into register *rd*. It computes its effective address by adding the zero-extended offset, scaled by 16, to the stack pointer, `x2`. It expands to `lq rd, offset[9:4](x2)`.

C.FLWSP is an RV32FC-only instruction that loads a single-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, `x2`. It expands to `flw rd, offset[7:2](x2)`.

C.FLDSP is an RV32DC/RV64DC-only instruction that loads a double-precision floating-point value from memory into floating-point register *rd*. It computes its effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, `x2`. It expands to `fld rd, offset[8:3](x2)`.

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

These instructions use the CSS format.

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

C.SDSP is an RV64C/RV128C-only instruction that stores a 64-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, x2. It expands to `sd rs2, offset[8:3](x2)`.

C.SQSP is an RV128C-only instruction that stores a 128-bit value in register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 16, to the stack pointer, x2. It expands to `sq rs2, offset[9:4](x2)`.

C.FSWSP is an RV32FC-only instruction that stores a single-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 4, to the stack pointer, x2. It expands to `fsw rs2, offset[7:2](x2)`.

C.FSDSP is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2* to memory. It computes an effective address by adding the *zero*-extended offset, scaled by 8, to the stack pointer, x2. It expands to `fsd rs2, offset[8:3](x2)`.

Register-Based Loads and Stores

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

These instructions use the CL format.

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

the *zero*-extended offset, scaled by 4, to the base address in register *rs1'*. It expands to `lw rd',` `offset[6:2](rs1')`.

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

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

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

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

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

These instructions use the CS format.

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

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

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

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

C.FSD is an RV32DC/RV64DC-only instruction that stores a double-precision floating-point value in floating-point register *rs2'* to memory. It computes an effective address by adding the *zero*-

extended offset, scaled by 8, to the base address in register *rs1'*. It expands to `fsd rs2'`, `offset[7:3](rs1')`.

14.4 Control Transfer Instructions

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

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

These instructions use the CJ format.

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

C.JAL is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (`pc+2`) to the link register, `x1`. C.JAL expands to `jal x1, offset[11:1]`.

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

These instructions use the CR format.

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

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

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

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

These instructions use the CB format.

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

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

14.5   Integer Computational Instructions

RVC provides several instructions for integer arithmetic and constant generation.

Integer Constant-Generation Instructions

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

15	13	12	11	7 6	2 1	0
funct3		imm[5]	rd	imm[4:0]	op	
3		1	5	5	2	
C.LI		imm[5]	dest≠0	imm[4:0]	C1	
C.LUI		nzimm[17]	dest≠{0, 2}	nzimm[16:12]	C1	

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

C.LUI loads the non-zero 6-bit immediate field into bits 17–12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI is only valid when *rd*≠{x0, x2}, and when the immediate is not equal to zero. C.LUI expands into `lui rd, nzimm[17:12]`.

Integer Register-Immediate Operations

These integer register-immediate operations are encoded in the CI format and perform operations on any non-x0 integer register and a 6-bit immediate. The immediate cannot be zero.

15	13	12	11	7 6	2 1	0
funct3		imm[5]	rd/rs1	imm[4:0]	op	
3		1	5	5	2	
C.ADDI		nzimm[5]	dest	nzimm[4:0]	C1	
C.ADDIW		imm[5]	dest≠0	imm[4:0]	C1	
C.ADDI16SP		nzimm[9]	2	nzimm[4|6|8:7|5]	C1	

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

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

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

15	13 12	5 4	2 1	0
funct3	imm	rd'	op	
3	8	3	2	
C.ADDI4SPN	zimm[5:4|9:6|2|3]	dest	C0	

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

15	13	12	11	7 6	2 1	0
funct3		shamt[5]	rd/rs1	shamt[4:0]	op	
3		1	5	5	2	
C.SLLI		shamt[5]	dest≠0	shamt[4:0]	C2	

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

15	13	12	11	10 9	7 6	2 1	0
funct3		shamt[5]	funct2	rd'/rs1'	shamt[4:0]	op	
3		1	2	3	5	2	
C.SRLI		shamt[5]	C.SRLI	dest	shamt[4:0]	C1	
C.SRAI		shamt[5]	C.SRAI	dest	shamt[4:0]	C1	

C.SRLI is a CB-format instruction that performs a logical right shift of the value in register rd' then writes the result to rd'. The shift amount is encoded in the *shamt* field, where *shamt[5]* must be zero for RV32C. For RV32C and RV64C, the shift amount must be non-zero. For RV128C, a shift amount of zero is used to encode a shift of 64. Furthermore, the shift amount is sign-extended for RV128C, and so the legal shift amounts are 1–31, 64, and 96–127. C.SRLI expands into `srli rd', rd', shamt[5:0]`, except for RV128C with `shamt=0`, which expands to `srli rd', rd', 64`.

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

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

15	13	12	11	10	9	7	6	2	1	0
funct3		imm[5]	funct2		rd'/rs1'		imm[4:0]		op	
3		1	2		3		5		2	
C.ANDI		imm[5]	C.ANDI		dest		imm[4:0]		C1	

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

Integer Register-Register Operations

15	12	11	7	6	2	1	0
funct4		rd/rs1		rs2		op	
4		5		5		2	
C.MV		dest≠0		src≠0		C0	
C.ADD		dest≠0		src≠0		C0	

These instructions use the CR format.

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

C.ADD adds the values in registers rd and $rs2$ and writes the result to register rd. C.ADD expands into `add rd, rd, rs2`.

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

These instructions use the CS format.

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

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

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

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

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

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

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

Defined Illegal Instruction

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

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

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

NOP Instruction

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

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

Breakpoint Instruction

15	12	11	2	1	0
funct4		0		op	
4		10		2	
C.EBREAK		0		C0	

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.

14.6 Usage of C Instructions in LR/SC Sequences

On implementations that support the C extension, compressed forms of the I instructions permitted inside LR/SC sequences can be used while retaining the guarantee of eventual success, as described in Section 6.2.

The implication is that any implementation that claims to support both the A and C extensions must ensure that LR/SC sequences containing valid C instructions will eventually complete.

14.7 RVC Instruction Set Listings

Table 14.3 shows a map of the major opcodes for RVC. Opcodes with the lower two bits set correspond to instructions wider than 16 bits, including those in the base ISAs. Several instructions are only valid for certain operands; when invalid, they are marked either *RES* to indicate that the opcode is reserved for future standard extensions; *NSE* to indicate that the opcode is reserved for non-standard extensions; or *HINT* to indicate that the opcode is reserved for future standard microarchitectural hints. Instructions marked *HINT* must execute as no-ops on implementations for which the hint has no effect.

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

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

Table 14.3: RVC opcode map

Tables 14.4–14.6 list the RVC instructions.

14.8 Instruction Compression Statistics

The following tables provide some data we used to guide the selection of instructions to include in RVC.

Table 14.7 lists the standard RVC instructions with the most frequent first, showing the individual contributions of those instructions to static code size and then the running total for three experiments: the SPEC benchmarks for both RV32C and RV64C for the Linux kernel. For RV32, RVC reduces static code size by 24.5% on Dhrystone and 30.9% on CoreMark. For RV64, it reduces static code size by 26.3% on SPECint, 25.8% on SPECfp, and 31.1% on the Linux kernel.

Table 14.8 ranks the RVC instructions by order of typical dynamic frequency. For RV32, RVC reduces dynamic bytes fetched by 29.2% on Dhrystone and 29.3% on CoreMark. For RV64, it reduces dynamic bytes fetched by 26.9% on SPECint, 22.4% on SPECfp, and 26.11% booting the Linux kernel.

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

Table 14.4: Instruction listing for RVC, Quadrant 0.

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

Table 14.5: Instruction listing for RVC, Quadrant 1.

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

Table 14.6: Instruction listing for RVC, Quadrant 2.

| | RV32GC | | | RV64GC | | |
Instruction	Dhry-stone	Core-Mark	SPEC 2006	SPEC 2006	Linux Kernel	Max
C.MV	1.78	5.03	4.06	3.62	5.00	5.03
C.LWSP	4.51	2.80	2.89	0.49	0.14	4.51
C.LDSP	—	—	—	3.20	4.44	4.44
C.SWSP	4.19	2.45	2.76	0.45	0.18	4.19
C.SDSP	—	—	—	2.75	3.79	3.79
C.LI	2.99	3.74	2.81	2.35	2.86	3.74
C.ADDI	2.16	3.28	1.87	1.19	0.95	3.28
C.ADD	0.51	1.64	1.94	2.28	0.91	2.28
C.LW	2.10	1.68	2.00	0.74	0.62	2.10
C.LD	—	—	—	1.14	2.09	2.09
C.J	0.32	1.71	1.63	0.97	1.53	1.71
C.SW	1.59	0.85	0.73	0.27	0.26	1.59
C.JR	1.52	1.16	0.49	0.44	1.05	1.52
C.BEQZ	0.38	1.14	0.76	0.55	1.24	1.24
C.SLLI	0.06	1.09	0.57	0.93	0.57	1.09
C.ADDI16SP	0.19	0.26	0.32	0.42	1.01	1.01
C.SRLI	0.00	0.81	0.05	0.12	0.31	0.81
C.BNEZ	0.19	0.53	0.53	0.32	0.80	0.80
C.SD	—	—	—	0.25	0.79	0.79
C.ADDIW	—	—	—	0.77	0.50	0.77
C.JAL	0.38	0.59	0.05	—	—	0.59
C.ADDI4SPN	0.57	0.37	0.45	0.50	0.30	0.57
C.LUI	0.32	0.37	0.44	0.56	0.52	0.56
C.SRAI	0.13	0.48	0.07	0.03	0.03	0.48
C.ANDI	0.00	0.42	0.20	0.07	0.35	0.42
C.FLD	0.00	0.00	0.16	0.39	0.00	0.39
C.FLDSP	0.00	0.02	0.20	0.31	0.00	0.31
C.FSDSP	0.13	0.09	0.15	0.26	0.00	0.26
C.SUB	0.25	0.09	0.13	0.06	0.11	0.25
C.AND	0.00	0.00	0.07	0.03	0.21	0.21
C.FSD	0.00	0.00	0.08	0.18	—	0.18
C.OR	0.06	0.18	0.09	0.04	0.14	0.18
C.JALR	0.13	0.07	0.17	0.10	0.14	0.17
C.ADDW	—	—	—	0.16	0.12	0.16
C.EBREAK	0.00	0.02	0.00	0.00	0.08	0.08
C.FLW	0.00	0.00	0.05	—	—	0.05
C.XOR	0.00	0.04	0.01	0.01	0.03	0.04
C.SUBW	—	—	—	0.04	0.03	0.04
C.FLWSP	0.00	0.00	0.03	—	—	0.03
C.FSW	0.00	0.00	0.02	—	—	0.02
C.FSWSP	0.00	0.00	0.02	—	—	0.02
Total	24.46	30.92	25.78	25.98	31.11	—

Table 14.7: RVC instructions in order of typical static frequency. The numbers in the table show the percentage savings in static code size attributable to each instruction. This list was generated using a compacting assembler for the output of the RISC-V GCC compiler, directed to use RV32GC for Dhrystone, CoreMark, and SPEC CPU2006, and RV64GC for SPEC CPU2006 and the Linux kernel, version 3.14.29. A dash means that instruction is not defined for this address size.

Instruction	RV32GC		RV64GC		Max
	Dhry-stone	Core-Mark	SPEC 2006	Linux Kernel	
C.ADDI	3.70	3.91	4.36	1.26	4.36
C.LW	4.15	3.89	1.09	0.87	4.15
C.MV	1.93	4.01	1.70	1.37	4.01
C.BNEZ	0.44	2.57	0.47	3.62	3.62
C.SW	3.55	1.62	0.32	0.68	3.55
C.LD	—	—	1.43	3.29	3.29
C.SWSP	3.26	0.32	0.20	0.03	3.26
C.LWSP	2.96	0.48	0.14	0.02	2.96
C.LI	2.22	1.47	0.81	2.73	2.73
C.ADD	2.07	2.69	2.64	1.84	2.69
C.SRLI	0.00	2.48	0.20	0.38	2.48
C.JR	2.07	0.34	0.46	0.42	2.07
C.FLD	0.00	0.00	1.63	0.00	1.63
C.SDSP	—	—	1.14	1.38	1.38
C.J	0.44	0.46	0.33	1.35	1.35
C.LDSP	—	—	1.34	1.31	1.34
C.ANDI	0.15	1.30	0.10	0.23	1.30
C.ADDIW	—	—	1.26	1.03	1.26
C.SLLI	0.15	1.10	1.24	0.89	1.24
C.SD	—	—	0.39	1.13	1.13
C.BEQZ	0.59	0.95	0.74	0.76	0.95
C.AND	0.00	0.00	0.21	0.75	0.75
C.SRAI	0.00	0.72	0.02	0.01	0.72
C.JAL	0.59	0.26	—	—	0.59
C.ADDI4SPN	0.44	0.16	0.07	0.05	0.44
C.FLDSP	0.00	0.00	0.40	0.00	0.40
C.ADDI16SP	0.13	0.18	0.28	0.38	0.38
C.FSD	0.00	0.00	0.29	0.00	0.29
C.FSDSP	0.00	0.00	0.25	0.00	0.25
C.ADDW	—	—	0.19	0.04	0.19
C.XOR	0.00	0.19	0.06	0.02	0.19
C.OR	0.15	0.08	0.05	0.04	0.15
C.SUB	0.15	0.03	0.05	0.04	0.15
C.LUI	0.02	0.06	0.09	0.10	0.10
C.JALR	0.00	0.05	0.05	0.03	0.05
C.SUBW	—	—	0.04	0.02	0.04
C.EBREAK	0.00	0.00	0.00	0.00	0.00
C.FLW	0.00	0.00	—	—	—
C.FLWSP	0.00	0.00	—	—	—
C.FSW	0.00	0.00	—	—	—
C.FSWSP	0.00	0.00	—	—	—
Total	29.18	29.29	24.03	26.11	—

Table 14.8: RVC instructions in order of typical dynamic frequency. The numbers in the table show the percentage savings in dynamic code size attributable to each instruction. This list was generated by executing CoreMark and Dhrystone compiled for RV32GC and SPEC CPU2006 compiled for RV64GC. For SPEC, we used the reference input set. The Linux boot includes the time to boot the kernel, then execute the init process, the shell, and the poweroff command.

14.9 Optimizing Register Save/Restore Code Size

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.

The standard RISC-V toolchain provides an alternative approach to reduce save/restore static code size even further in exchange for reduced performance. Instead of inlining the register save/restore code in each function, register save code is replaced with a jump-and-link instruction to call a subroutine to copy registers to the stack then return to the function. Register restore code is replaced with a jump to a routine that restores registers from the stack then jumps to the restored return address.

Figure 14.1 shows the impact on static code size and dynamic instruction count of these routines when naïvely applied to all functions in the SPEC CPU2006 benchmarks. On average, code size is reduced by 4% in exchange for a 3% increase in dynamic instruction count.

The inline save/restore code is replaced with calls to the save/restore subroutines when the `-Os` flag (reduce code size) is passed to `gcc`.

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

- *These instructions complicate processor implementations.*

- *For virtual memory systems, some data accesses could be resident in physical memory and some could not, which requires a new restart mechanism for partially executed instructions.*

- *Unlike the rest of the RVC instructions, there is no IFD equivalent to Load Multiple and Store Multiple.*

- *Unlike the rest of the RVC instructions, the compiler would have to be aware of these instructions to both generate the instructions and to allocate registers in an order to maximize the chances of the them being saved and stored, since they would be saved and restored in sequential order.*

- *Simple microarchitectural implementations will constrain how other instructions can be scheduled around the load and store multiple instructions, leading to a potential performance loss.*

- *The desire for sequential register allocation might conflict with the featured registers selected for the CIW, CL, CS, and CB formats.*

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.
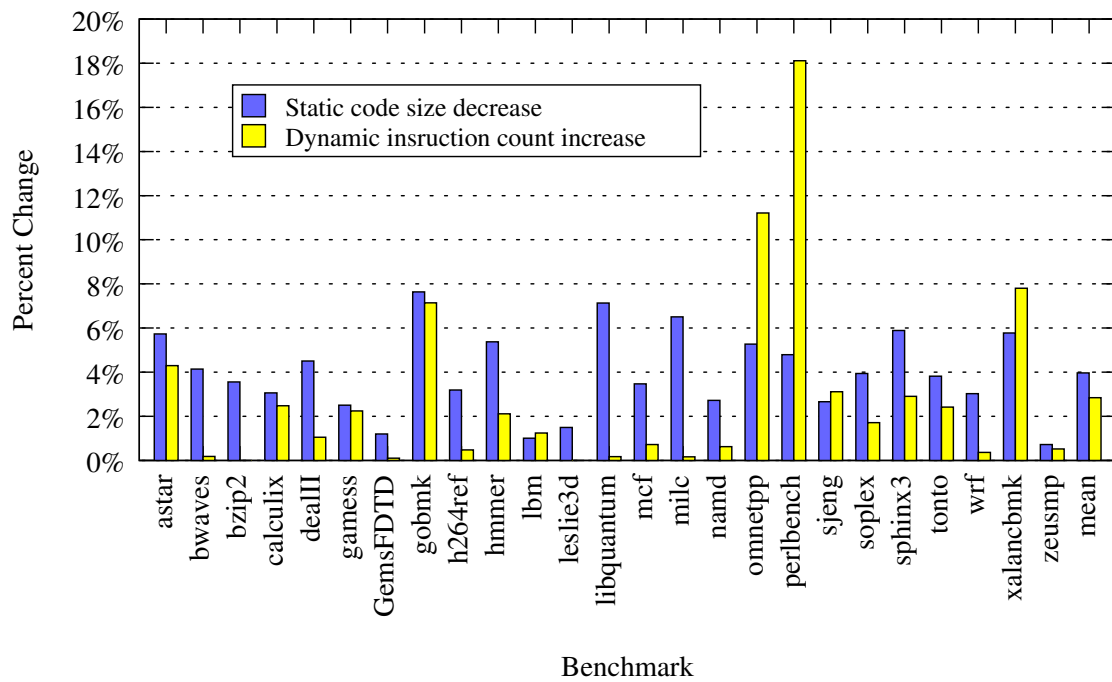
Figure 14.1: Impact on static code size and dynamic instruction count of compressed function prologue and epilogue subroutines.

Chapter 15

"V" Standard Extension for Vector Operations, Version 0.0

This chapter is a placeholder for a future standard extension to provide vector instructions.

Chapter 16

"B" Standard Extension for Bit Manipulation, Version 0.0

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

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

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

Chapter 17

"T" Standard Extension for Transactional Memory, Version 0.0

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

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

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

Chapter 18

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

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

> *Packed-SIMD extensions, first introduced with the Lincoln Labs TX-2 [5], have become a popular way to provide higher throughput on data-parallel codes. Earlier commercial microprocessor implementations include the Intel i860, HP PA-RISC MAX [15], SPARC VIS [24], MIPS MDMX [8], PowerPC AltiVec [4], Intel x86 MMX/SSE [19, 21], while recent designs include Intel x86 AVX [16] and ARM Neon [7]. We describe a standard framework for adding packed-SIMD in this chapter, but are not actively working on such a design. In our opinion, packed-SIMD designs represent a reasonable design point when reusing existing wide datapath resources, but if significant additional resources are to be devoted to data-parallel execution then designs based on traditional vector architectures are a better choice and should use the V extension.*

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

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

The existing floating-point load and store instructions are used to load and store various-sized words from memory to the `f` registers. The base ISA supports 32-bit and 64-bit loads and stores, but the LOAD-FP and STORE-FP instruction encodings allows 8 different widths to be encoded as shown

in Table 18.1. When used with packed-SIMD operations, it is desirable to support non-naturally aligned loads and stores in hardware.

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

Table 18.1: LOAD-FP and STORE-FP width encoding.

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

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

Chapter 19

RV128I Base Integer Instruction Set, Version 1.7

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

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

> *The primary reason to extend integer register width is to support larger address spaces. It is not clear when a flat address space larger than 64 bits will be required. At the time of writing, the fastest supercomputer in the world as measured by the Top500 benchmark had over 1 PB of DRAM, and would require over 50 bits of address space if all the DRAM resided in a single address space. Some warehouse-scale computers already contain even larger quantities of DRAM, and new dense solid-state non-volatile memories and fast interconnect technologies might drive a demand for even larger memory spaces. Exascale systems research is targeting 100 PB memory systems, which occupy 57 bits of address space. At historic rates of growth, it is possible that greater than 64 bits of address space might be required before 2030.*
>
> *History suggests that whenever it becomes clear that more than 64 bits of address space is needed, architects will repeat intensive debates about alternatives to extending the address space, including segmentation, 96-bit address spaces, and software workarounds, until, finally, flat 128-bit address spaces will be adopted as the simplest and best solution.*
>
> *We have not frozen the RV128 spec at this time, as there might be need to evolve the design based on actual usage of 128-bit address spaces.*

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

Shifts by an immediate (SLLI/SRLI/SRAI) are now encoded using the low 7 bits of the I-immediate, and variable shifts (SLL/SRL/SRA) use the low 7 bits of the shift amount source register.

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

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

Chapter 20

Calling Convention

This chapter describes the C compiler standards for RV32 and RV64 programs and two calling conventions: the convention for the base ISA plus standard general extensions (RV32G/RV64G), and the soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I).

Implementations with ISA extensions might require extended calling conventions.

20.1 C Datatypes and Alignment

Table 20.1 summarizes the datatypes natively supported by RISC-V C programs. In both RV32 and RV64 C compilers, the C type `int` is 32 bits wide. `long`s and pointers, on the other hand, are both as wide as a integer register, so in RV32, both are 32 bits wide, while in RV64, both are 64 bits wide. Equivalently, RV32 employs an ILP32 integer model, while RV64 is LP64. In both RV32 and RV64, the C type `long long` is a 64-bit integer, `float` is a 32-bit IEEE 754-2008 floating-point number, `double` is a 64-bit IEEE 754-2008 floating-point number, and `long double` is a 128-bit IEEE floating-point number.

The C types `char` and `unsigned char` are 8-bit unsigned integers and are zero-extended when stored in a RISC-V integer register. `unsigned short` is a 16-bit unsigned integer and is zero-extended when stored in a RISC-V integer register. `signed char` is an 8-bit signed integer and is sign-extended when stored in a RISC-V integer register, i.e. bits (XLEN-1)..7 are all equal. `short` is a 16-bit signed integer and is sign-extended when stored in a register.

In RV64, 32-bit types, such as `int`, are stored in integer registers as proper sign extensions of their 32-bit values; that is, bits 63..31 are all equal. This restriction holds even for unsigned 32-bit types.

The RV32 and RV64 C compiler and compliant software keep all of the above datatypes naturally aligned when stored in memory.

C type	Description	Bytes in RV32	Bytes in RV64
`char`	Character value/byte	1	1
`short`	Short integer	2	2
`int`	Integer	4	4
`long`	Long integer	4	8
`long long`	Long long integer	8	8
`void*`	Pointer	4	8
`float`	Single-precision float	4	4
`double`	Double-precision float	8	8
`long double`	Extended-precision float	16	16

Table 20.1: C compiler datatypes for base RISC-V ISA.

20.2   RVG Calling Convention

The RISC-V calling convention passes arguments in registers when possible. Up to eight integer registers, a0–a7, and up to eight floating-point registers, fa0–fa7, are used for this purpose.

If the arguments to a function are conceptualized as fields of a C struct, each with pointer alignment, the argument registers are a shadow of the first eight pointer-words of that struct. If argument $i < 8$ is a floating-point type, it is passed in floating-point register fa$i$; otherwise, it is passed in integer register ai. However, floating-point arguments that are part of unions or array fields of structures are passed in integer registers. Additionally, floating-point arguments to variadic functions (except those that are explicitly named in the parameter list) are passed in integer registers.

Arguments smaller than a pointer-word are passed in the least-significant bits of argument registers. Correspondingly, sub-pointer-word arguments passed on the stack appear in the lower addresses of a pointer-word, since RISC-V has a little-endian memory system.

When primitive arguments twice the size of a pointer-word are passed on the stack, they are naturally aligned. When they are passed in the integer registers, they reside in an aligned even-odd register pair, with the even register holding the least-significant bits. In RV32, for example, the function void foo(int, long long) is passed its first argument in a0 and its second in a2 and a3. Nothing is passed in a1.

Arguments more than twice the size of a pointer-word are passed by reference.

The portion of the conceptual struct that is not passed in argument registers is passed on the stack. The stack pointer sp points to the first argument not passed in a register.

Values are returned from functions in integer registers a0 and a1 and floating-point registers fa0 and fa1. Floating-point values are returned in floating-point registers only if they are primitives or members of a struct consisting of only one or two floating-point values. Other return values that fit into two pointer-words are returned in a0 and a1. Larger return values are passed entirely in memory; the caller allocates this memory region and passes a pointer to it as an implicit first parameter to the callee.

In the standard RISC-V calling convention, the stack grows downward and the stack pointer is always kept 16-byte aligned.

In addition to the argument and return value registers, seven integer registers `t0`–`t6` and twelve floating-point registers `ft0`–`ft11` are temporary registers that are volatile across calls and must be saved by the caller if later used. Twelve integer registers `s0`–`s11` and twelve floating-point registers `fs0`–`fs11` are preserved across calls and must be saved by the callee if used. Table 20.2 indicates the role of each integer and floating-point register in the calling convention.

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

Table 20.2: RISC-V calling convention register usage.

20.3 Soft-Float Calling Convention

The soft-float calling convention is intended for use on RV32 and RV64 implementations that lack floating-point hardware. It avoids all use of instructions in the F, D, and Q standard extensions, and hence the `f` registers.

Integral arguments are passed and returned in the same manner as the RVG convention, and the stack discipline is the same except that the stack is only kept aligned to XLEN/8-byte boundaries (e.g., four-byte alignment for RV32I).

As floating-point data on the stack will be accessed using integer load and store instructions, there is no incentive to maintain stack alignment at a coarse granularity in the soft-float calling convention. The reduced stack alignment saves space in the memory-constrained systems that might commonly use soft floating-point.

Floating-point arguments are passed and returned in integer registers, using the rules for integer arguments of the same size. In RV32, for example, the function `double foo(int, double, long double)` is passed its first argument in `a0`, its second argument in `a2` and `a3`, and its third argument by reference via `a4`; its result is returned in `a0` and `a1`. In RV64, the arguments are passed in `a0`, `a1`, and the `a2-a3` pair, and the result is returned in `a0`.

The dynamic rounding mode and accrued exception flags are accessed through the routines provided by the C99 header `fenv.h`.

20.4   RV32E Calling Convention

RV32E uses a subset of the Soft-Float calling convention. As only 16 integer registers `x0`–`x15` are present, there are only six argument registers (`x10`–`x15`), two saved registers (`x8`–`x9`), and three temporary registers (`x5`–`x7`). The stack is kept aligned on a four-byte boundary.

Chapter 21

RISC-V Assembly Programmer's Handbook

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

Table 21.1 contains a list of standard RISC-V pseudoinstructions.

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

Table 21.1: RISC-V pseudoinstructions.

Chapter 22

History and Acknowledgments

22.1   History from Revision 1.0 of ISA manual

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

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

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

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

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

Based on experience with all these previous projects, the RISC-V ISA definition was begun in Summer 2010. An initial version of the RISC-V 32-bit instruction subset was used in the UC Berkeley Fall 2010 CS250 VLSI Systems Design class, with Yunsup Lee as TA. RISC-V is a clean break from the earlier MIPS-inspired designs. John Hauser contributed to the floating-point ISA definition.

22.2   History from Revision 2.0 of ISA manual

Multiple implementations of RISC-V processors have been completed, including several silicon fabrications, as shown in Figure 22.1.

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

Table 22.1: Fabricated RISC-V testchips.

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

Subsequently, a clean microarchitecture for an in-order decoupled RV64 core was developed by Andrew Waterman, Rimas Avižienis, and Yunsup Lee, advised by Krste Asanović, and, continuing the railway theme, was codenamed "Rocket" after George Stephenson's successful steam locomotive design. Rocket was written in Chisel, a new hardware design language developed at UC Berkeley. The IEEE floating-point units used in Rocket were developed by John Hauser, Andrew Waterman, and Brian Richards. Rocket has since been refined and developed further, and has been fabricated two more times in 28 nm FDSOI (Raven-2, Raven-3), and five times in IBM 45 nm SOI technology (EOS14, EOS16, EOS18, EOS20, EOS22) for a photonics project. Work is ongoing to make the Rocket design available as a parameterized RISC-V processor generator.

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

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

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

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

Andrew Waterman completed a Master's thesis with a preliminary design of the RISC-V compressed instruction set [28].

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

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

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

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

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

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

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

Acknowledgments

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

22.3 History for Revision 2.1

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

Acknowledgments

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

22.4 Funding

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

Bibliography

[1] IEEE standard for a 32-bit microprocessor. IEEE Std. 1754-1994, 1994.

[2] G. M. Amdahl, G. A. Blaauw, and Jr. F. P. Brooks. Architecture of the IBM System/360. *IBM Journal of R. & D.*, 8(2), 1964.

[3] Werner Buchholz, editor. *Planning a computer system: Project Stretch*. McGraw-Hill Book Company, 1962.

[4] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85 –95, 2000.

[5] John M. Frankovich and H. Philip Peterson. A functional description of the Lincoln TX-2 computer. In *Western Joint Computer Conference*, Los Angeles, CA, February 1957.

[6] 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.

[7] J. Goodacre and A.N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42 – 50, 2005.

[8] Linley Gwennap. Digital, MIPS add multimedia extensions. Microprocessor Report, 1996.

[9] Timothy H. Heil and James E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.

[10] ANSI/IEEE Std 754-2008, IEEE standard for floating-point arithmetic, 2008.

[11] Manolis G.H. Katevenis, Robert W. Sherburne, Jr., David A. Patterson, and Carlo H. Séquin. The RISC II micro-architecture. In *Proceedings VLSI 83 Conference*, August 1983.

[12] Hyesoon Kim, Onur Mutlu, Jared Stark, and Yale N. Patt. Wish branches: Combining conditional branching and predication for adaptive predicated execution. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 43–54, 2005.

[13] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, Washington, DC, USA, 1998.

[14] David D. Lee, Shing I. Kong, Mark D. Hill, George S. Taylor, David A. Hodges, Randy H. Katz, and David A. Patterson. A VLSI chip set for a multiprocessor workstation–Part I: An RISC microprocessor with coprocessor interface and support for symbolic processing. *IEEE JSSC*, 24(6):1688–1698, December 1989.

[15] R.B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51 –59, August 1996.

[16] Chris Lomont. Introduction to Intel Advanced Vector Extensions. Intel White Paper, 2011.

[17] OpenCores. OpenRISC 1000 architecture manual, architecture version 1.0, December 2012.

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

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

[20] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305. IEEE Computer Society, 2001.

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

[22] Balaram Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1–1, 2011.

[23] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*, AFIPS '64 (Fall, part II), pages 33–40, 1965.

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

[25] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[26] 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.

[27] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *ISCA*, pages 188–197, Ann Arbor, MI, 1984.

[28] Andrew Waterman. Improving energy efficiency and reducing code size with RISC-V compressed. Master's thesis, University of California, Berkeley, 2011.

[29] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[30] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. The RISC-V instruction set manual, Volume I: Base user-level ISA version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.