Creating a custom processor with RISC-V

Richard Quinnell - February 14, 2019

Title-3

With its blend of open-source freedoms with the benefits of standardization, the RISC-V (*risk-five*) Foundation is attracting widespread industry interest. Its core specifications are stable and on the cusp of ratification, soft- and hard CPU cores along with chips, development boards, and tools are commercially available, and major companies have started adopting RISC-V to replace their custom architectures. A key feature in the architecture's appeal is that CPU developers can adapt RISC-V functionality to their needs without sacrificing the applicability of tools and libraries created for the base standard. The key to that adaptation lies in understanding RISC-V's modular instruction set architecture.

RISC-V started as the fifth iteration of reduced instruction set computing (RISC) design efforts at UC Berkeley, but quickly evolved from academic research to a movement seeking to redefine the electronics industry's processing hardware design approach. Currently, system developers either must choose a proprietary CPU architecture, often optimized to a specific application space, or design their own CPU architecture. By pursuing their own design, however, developers give up the extensive support ecosystems that established CPUs have developed. There is a compromise: adapting a proprietary CPU architecture to gain customization while retaining much of the support ecosystem. This compromise, unfortunately, is impractical for many design teams due to high architecture licensing fees for proprietary architectures.

Editor's Note: This article is part of an AspenCore Special Project, a collection of interrelated articles that explores hardware, software, and business issues surrounding RISC-V technology. RISC-V is an open instruction set architecture that supports customization on top of a standard core so that developers can develop a support ecosystem easily extended to accommodate user-specific extensions.

The RISC-V initiative seeks to offer designers an alternative that permits customization and innovation while retaining many benefits of standardization. To do so, the RISC-V Foundation maintains and drives community development of the modular, open source, RISC-V processor instruction set architecture (ISA), which aims to meet application needs spanning embedded systems to server farms and beyond. The architecture's specifications are free to download, and developers are free to implement designs based on the ISA without paying a license fee. Nor are they obligated to make their designs available to others, as with some open source initiatives. It is the ISA that is

open source; individual designs, hardware architectures, and customizations can remain proprietary if developers wish.

The initiative has gained considerable momentum. There are now RISC-V chips and cores available, both commercial and open source. Companies like SiFive, GreenWaves Technologies, and Microsemi have development boards for their RISC-V implementations. Development tools, software libraries, and operating system ports (including Linux) are all part of the current RISC-V support ecosystem. Leveraging all this support for a custom design, though, begins with a close look at the RISC-V ISA's structure.

Base specifications

Two key documents define the RISC-V ISA: The User-level ISA Specification and the Privileged ISA Specification. Within these are the definitions of both base requirements and numerous standardized, modular extensions. The standard extensions are modular in that implementing any given standard extension in a CPU design will not interfere with the implementation of any other standard extensions. Some extensions may build on others, however, requiring that base extension be implemented as part of the desired extension.

The overall design is register-based, calling for 31 general-purpose registers where all operations take place, with load-and-store access to the general memory space. Instruction sets have been defined for 32-bit, 64-bit, and 128-bit address spaces, with an additional, reduced-register-count, 32-bit instruction set defined to specifically target smaller gate count implementations for embedded system designs. Apart from the presence of additional instructions to manipulate longer word lengths, and register size matching the address space, the instruction coding for these variations is all the same.

Figure 1 shows in schematic form how the core specifications and standard extensions interact. Many of the specifications are now frozen by the RISC-V Foundation, ensuring that implementations based on them will remain valid as the ISA evolves. This ensures that software written today will run on similar RISC-V cores forever. Some extensions are still in draft mode and so are subject to change. A few are reserved, i.e., placeholders awaiting future development. The 32- and 64-bit base integer ISAs are frozen, for instance, with the 128-bit and embedded variations still in draft form.

The base integer ISA (I) is the foundation on which all else is built and must be present in any implementation. In addition to the base integer ISA, all standard RISC-V implementations must include at least the machine-level portion of the Privileged ISA; the supervisor-level (S) and hypervisor portions are standard extensions. However, the Privileged ISA is defined in such a way that developers can implement a custom form of privileged code execution without affecting the base integer ISA, should they wish.

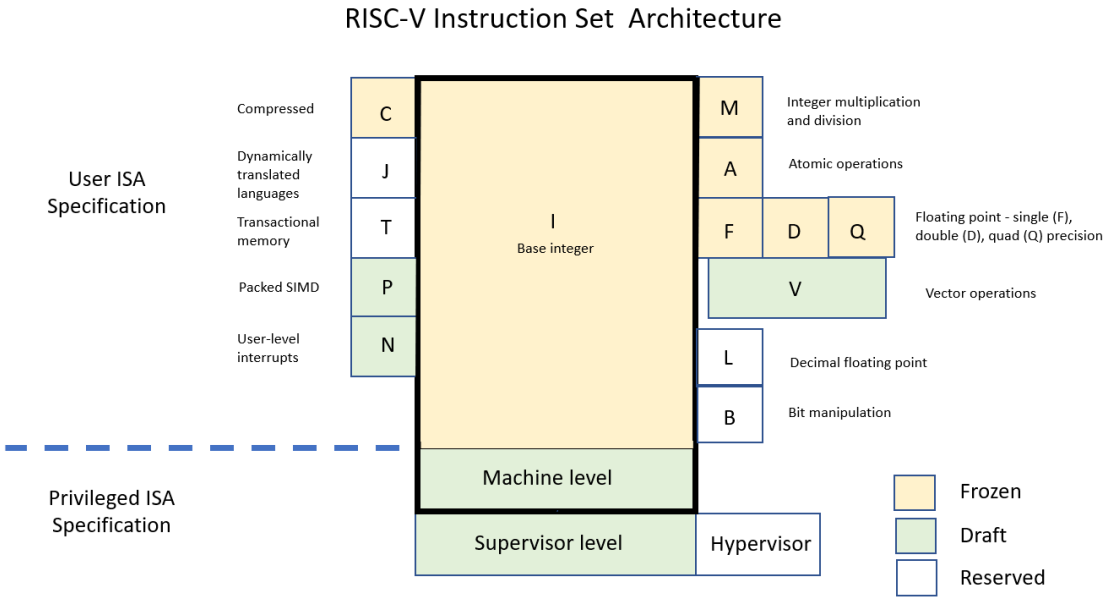


Figure 1 The RISC-V ISA forms a modular collection of instructions that can be implemented in a CPU design without interfering with one another.

Enhancing base functionality with standard extensions

The base integer (I) and machine-level privileged ISAs provide all the functionality a basic, general-purpose CPU requires. Developers can augment this base capability, though, by adding extensions to the ISA. Custom extensions are always possible, but there are standard extensions that the technical task groups in the RISC-V Foundation manage, having determined they have broad appeal to the design community and that their instructions do not conflict with other standard extensions. Developers are thus able to freely include any of the standard extensions they require in their design, without concern for conflicts in the instruction coding. These standard extensions include:

- M – Instructions that multiply and divide values held in two integer registers (frozen)
- A – Instructions that atomically read-modify-write memory to support synchronization (frozen)
- F, D, and Q – Instructions for (F) single-, (D) double-, and (Q) quad-precision floating-point computations compliant with the IEEE 754-2008 arithmetic standard. Each precision's extension depends on the lower-precision extension being present. (frozen)
- G – An implementation that includes the base integer specification (I) along with the M, A, F, and D standard extensions is so popular that the Foundation has defined the collection as G and has set the G configuration as the standard target of compiler toolchains under development. (frozen)
- V – Instructions to add vector instructions to the floating-point extensions (draft)
- L – Instructions for decimal floating-point calculations (reserved)
- B – Instructions for bit-level manipulation (reserved)
- N – Instructions that handle user-level interrupts (draft)
- P – An extension to support packed single-instruction, multiple-data instructions (reserved)
- T – Instructions to support transactional memory operations (reserved)
- J – An extension to support use of dynamically translated languages (reserved)
- C – Support for compressed instruction execution. The base integer (I) specification calls for instruction words to be 32-bits long and aligned on 32-bit boundaries in memory. Implementing the C standard extension provides 16-bit encodings of common operations and allows CPU designs to work with alignments on freely mixed 32- and 16-bit boundaries, resulting in a 25% to 30% reduction in code size. It can be implemented in any of the base integer bit widths and with any of the other standard extensions. (frozen)

- S – The privileged ISA's supervisor-level extension (draft)

Privileged ISA

Like the user-level ISA, the privileged ISA offers designers the ability to choose how much complexity to include. The specification defines two ISA sets: machine level and supervisor level, as well as reserving a placeholder for instructions to support hypervisor functionality. These instruction sets, in turn, allow developers to support up to three privilege levels at which code can operate. A single privilege level (machine level) means that all the code running has full access to the system resources, such as a single application running in a simple embedded system. Having two privilege levels, which requires both machine and supervisor ISA instructions, supports the isolation of some system resources from the application code for enhanced software security and to allow an operating system to multiple concurrent applications. Three privilege levels will support Unix-like operating systems, hypervisors, and the like.

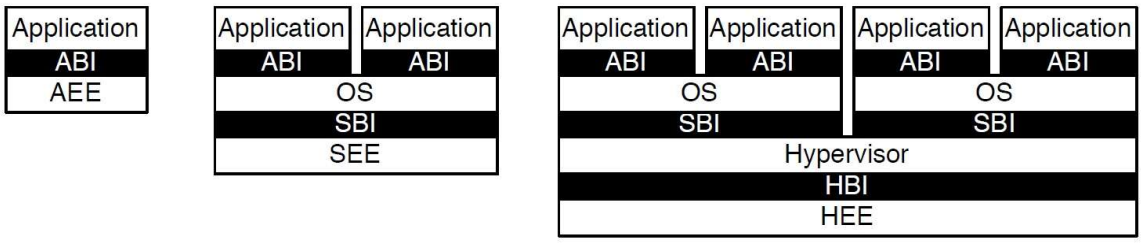


Figure 2 The two privileged ISA levels together can support many software configurations, including a simple application execution environment (AEE), multiple applications on an operating system with a supervisor execution environment (SEE), and multiple operating systems with a hypervisor. (**Source**: RISC-V Foundation)

With all the possible combinations of standard extensions and privilege levels, the simple designation "RISC-V" is inadequate to characterize an actual hardware implementation of the ISA. To clarify what instructions programmers can access when they are adopting a given hardware implementation, the Foundation has devised a core naming nomenclature. The name has three parts: the base specification used (RV32I, RV64I, etc.), the standard extensions added (M, F, A, etc.), and the version numbers (1p2 for version 1.2, etc.) for each element, as shown in **Figure 3**. In many cases the version numbers can be left off for simplicity (RV32IC, RV64G, etc.).

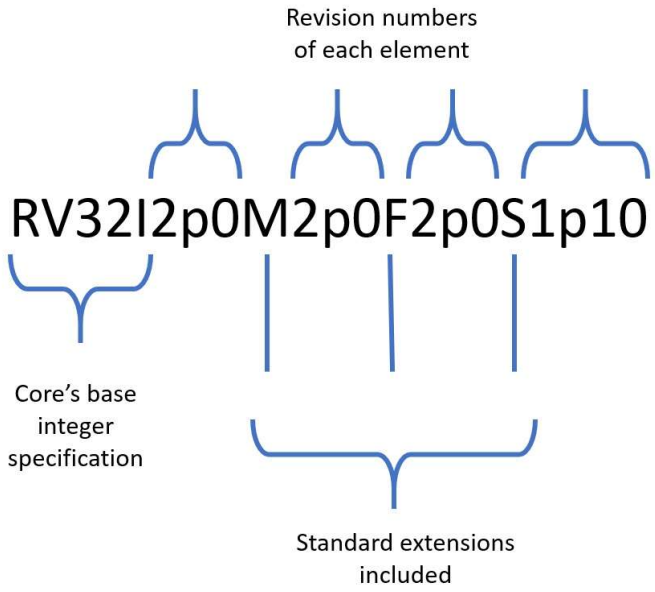


Figure 3 The name of a RISC-V implementation encodes a full description of the instruction set it supports.

The naming convention allows for identifying customized extensions, as well. Extensions to the base integer ISA employ a name the developer choses and have the form X*name*, with version number as appropriate. If the extension involves the supervisor level of the privileged ISA, the form is SX*name*.

Going beyond standard extensions

Even with the design flexibility that the modular standard extensions provide, the RISC-V standards do not offer many of the instruction enhancements that developers might desire. Consider, for example, an application that frequently requires the rounded-down averaging -- (r1+r2)/2 -- of two 16-bit integers, such as two ADC measurements. Using the base integer ISA, the desired averaging computation would require executing two instructions: integer addition and arithmetic shift right (which effectively performs an integer divide by two, rounded down). A custom instruction that performs both operations in a single step could therefore speed the application's software execution. The RISC-V ISA easily allows the addition of such an instruction if you adopt the ISA's standard instruction formats.

The RV32I base instruction set follows four basic formats, as shown in **Figure 4**. R-type instructions take values from two source registers (*rs1* and *rs2*) and combine them in some way (add, XOR, etc.) to form a value to be stored in a third in a destination register (*rd*). I-type instructions take a source value (*rs1*) and a 12-bit value encoded in the instruction itself (*imm*) to combine and store in a destination register (*rd*). Load instructions use the I-type format, combining the sourced and immediate values to determine a memory address and have its contents transferred to the destination register. S-type instructions take values from one source register to store in memory at the address that the second register's contents and the immediate value, when combined, point to. The B-type variant of the S-type instruction has the same format but uses the two values to calculate a conditional branch instruction address. U-type instructions allow use of an immediate value larger than 12 bits, while the J-type variant uses the immediate value to perform an unconditional jump.

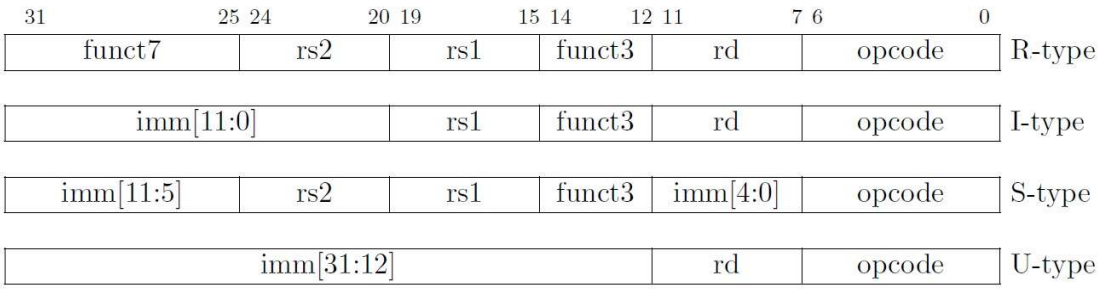


Figure 4 The four basic formats of RISC-V instructions. (Source: RISC-V Foundation)

The definition of these instruction formats provides several clues as to how a developer might easily add a custom instruction to the mix. All instructions include a 7-bit opcode in the lower bits, and all but the U-type format have a function code (*funct3*) at bits 12 to 14. The R-type instructions have a second function code (*funct7*) at bits 25 to 31. When a destination register address is involved it is always at bits 7 to 11, the first source register address is always at bits 15 to 19, and the other source register is at bits 20 to 24. This consistency in placement of codes and register pointers means that if you can map your new instruction to one of these base formats and you have a compliant RISC-V design to start from, the hardware design for implementing the instruction is almost already done. Most of the new operation – like instruction decoding and register data access – is in the existing design, and you can tap into it.

Extension example

To illustrate using the averaging example, consider the flow diagram in **Figure 5**. The existing RV32I ADD and SUB instructions for integers follow the R-type format and have the exact same opcode (0110011) and *funct3* (000) code values, differing only in their *funct7* code (0000000 versus 0100000). No other RV32I instruction uses that opcode and *funct3* combination, leaving more than 120 possible *funct7* values available for coding new instructions without stepping on any standard instruction. A possible implementation of these standard instructions shows the values from the two source registers going to an arithmetic unit which (depending on bit instruction 30) either adds or subtracts the values, delivering the result to the destination register.

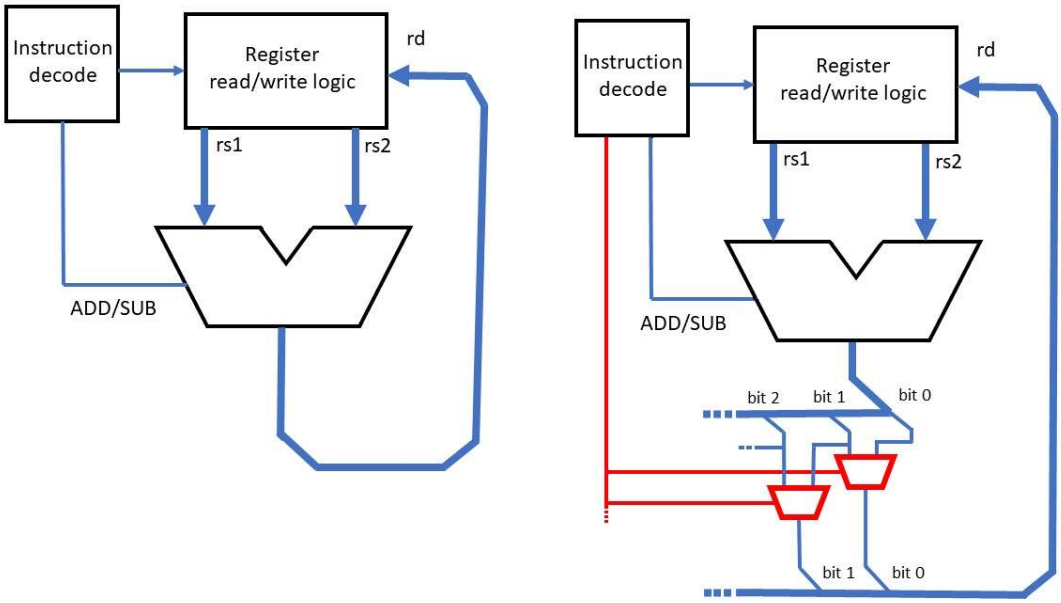


Figure 5 Modification of a standard RISV-V implementation to add two new instructions

Adding the new averaging instruction requires only minimal changes to this hardware. It might be done, for instance, by extracting instruction bit 25 from the instruction decoding logic and using it to command a set of new multiplexers inserted after the arithmetic unit. This bit causes the multiplexers to shift the otherwise standard ADD or SUB result one-bit right before delivering it to the destination register. The design will now implement ADD and SUB as well as two new instructions: average [(*rs1 + rs2*)/2] and split-the-difference [(*rs1-rs2*)/2] for just a handful of additional gates. Such a custom extension that works around existing instruction coding known as a brownfield extension.

Instruction expansion opportunities abound

RISC-V offers many opportunities for brownfield extensions. The RV32I base ISA defines its 47 different instructions using only 11 of the 128 (7-bit) major opcodes available, by leveraging the 3-bit *funct3* and 7-bit *funct7* values to define minor variations on major instruction types. Most of the standard extensions and the longer base integer variations (RV64I, RV128I) require only a few additional major opcodes. That leaves plenty of room for encoding new brownfield extension instructions. Only the Compressed Instruction (C) standard extension adds a significant number of unique instruction codes to account for its many variants in instruction length. But this, too, has been implemented in a way that minimizes opcode demand.

The chart below (**Figure 6**) shows the undefined major opcodes available in 32-, 64-, and 128-bit wide base integer implementations with the G standard extension, expected as the most common configurations that will be available for developers to build upon. The Foundation will avoid using some of these in any future standard extensions, helping ensure developers that their designs will be compatible with such extensions. Others are reserved for use by future standard extensions as well as for use by the anticipated 128-bit base integer standard but are available to developers not concerned with avoiding incompatibilities that might arise in that future.

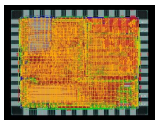
Figure 6 Map of available RISC-V major opcode

For developers who wish to create customized extensions to the base integer (I) specification without regard to all the standard extensions, the opportunities are even more abundant. For instance, the RISC-V ISA defines all its base integer (I) and most standard extension instructions with encodings that have the two least significant bits (LSBs) set to x11. Only the Compressed (C) standard extension defines instructions that have these bits set to x00, x01, or x10. A developer not requiring the C standard extension is then free to define instructions with those LSB bit patterns any way they desire. This gives them three, 30-bit instruction encoding spaces in which to play without compromising the base integer ISA or any of the other standard extensions.

A major goal of the RISC-V initiative is to allow innovation and customization without creating undue fragmentation of the growing ecosystem. Developers who adhere to the guidelines and requirements of the ISA as they expand upon the base architecture can help ensure that goal is met.

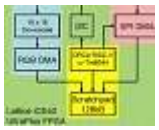
Rich Quinnell is an engineer, writer, and Global Managing Editor for the AspenCore Network.

For more in-depth insight into technical and business aspects of RISC-V, check out these other articles in this AspenCore Special Project.



[RISC-V on the Verge of Broad Adoption]

RISC-V's open ISA aims to redefine how processors get designed by enabling an ecosystem that supports both standardized and customized CPUs spanning a broad application space. Solidifying specifications, increasing adoption, and growing software and development support are helping clear the path to that goal.



[Introducing RISC-V and RISC-V tools]

It seems like everyone is talking about RISC-V processors these days, but what exactly is RISC-V and what tools are available to designers?

[RISC-V Climbs Software Mountain]

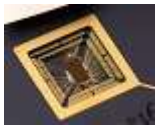
The open-source architecture faces a long road through software standards from its beachhead as an SoC controller to use as a host processor.

[Can Arm Survive the RISC-V challenge?]

Arm offers limited flexibility compared to RISC-V or MIPS. No one wants to spend months negotiating license terms under today's cost and time-to-market pressures.

[SiFive Sees Big Year for RISC-V]

This startup expects many design wins and new players.



[Open Source Hardware Benefits Procurement Practices]

The advent of processor options based the RISC-V ISA may delight electronics engineers and designers, but open-source hardware presents a number of opportunities for enhanced supply chain and procurement efforts as well.

[Can MIPS Leapfrog RISC-V?]

MIPS will become a bona fide open-source ISA. But given that MIPS will offer "commercial-ready" instruction sets with "industrial-strength" architecture, the hardware developers MIPS would attract are bigger and more mature companies, including current Arm licensees, according to Wave.

Related articles:

- Open source reaches processor core

- CPU selection in embedded systems
- AI hardware acceleration needs careful requirements planning
- Afternoon diversion: Design your own microprocessor