Design of the RISC-V Instruction Set Architecture

Andrew Waterman

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-1
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

January 3, 2016

Design of the RISC-V Instruction Set Architecture

by

Andrew Shell Waterman

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David Patterson, Chair
Professor Krste Asanović
Associate Professor Per-Olof Persson

Spring 2016

Design of the RISC-V Instruction Set Architecture

Abstract

Design of the RISC-V Instruction Set Architecture

by

Andrew Shell Waterman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Patterson, Chair

The hardware-software interface, embodied in the instruction set architecture (ISA), is arguably the most important interface in a computer system. Yet, in contrast to nearly all other interfaces in a modern computer system, all commercially popular ISAs are proprietary. A free and open ISA standard has the potential to increase innovation in microprocessor design, reduce computer system cost, and, as Moore's law wanes, ease the transition to more specialized computational devices.

In this dissertation, I present the RISC-V instruction set architecture. RISC-V is a free and open ISA that, with three decades of hindsight, builds and improves upon the original Reduced Instruction Set Computer (RISC) architectures. It is structured as a small base ISA with a variety of optional extensions. The base ISA is very simple, making RISC-V suitable for research and education, but complete enough to be a suitable ISA for inexpensive, low-power embedded devices. The optional extensions form a more powerful ISA for general-purpose and high-performance computing. I also present and evaluate a new RISC-V ISA extension for reduced code size, which makes RISC-V more compact than all popular 64-bit ISAs.

Contents

Contents **i**

List of Figures **iii**

List of Tables **v**

1 Introduction **1**

2 Why Develop a New Instruction Set? **3**
2.1 MIPS . . . 3
2.2 SPARC . . . 5
2.3 Alpha . . . 7
2.4 ARMv7 . . . 8
2.5 ARMv8 . . . 9
2.6 OpenRISC . . . 11
2.7 80x86 . . . 11
2.8 Summary . . . 13

3 The RISC-V Base Instruction Set Architecture **15**
3.1 The RV32I Base ISA . . . 16
3.2 The RV32E Base ISA . . . 27
3.3 The RV64I Base ISA . . . 28
3.4 The RV128I Base ISA . . . 30
3.5 Discussion . . . 30

4 The RISC-V Standard Extensions **32**
4.1 Integer Multiplication and Division . . . 32
4.2 Multiprocessor Synchronization . . . 34
4.3 Single-Precision Floating-Point . . . 38
4.4 Double-Precision Floating-Point . . . 45
4.5 Discussion . . . 46

5 The RISC-V Compressed ISA Extension **48**

5.1 Background . . . 48
5.2 Implications for the Base ISA . . . 50
5.3 RVC Design Philosophy . . . 51
5.4 The RVC Extension . . . 55
5.5 Evaluation . . . 59
5.6 The Load-Multiple and Store-Multiple Instructions . . . 66
5.7 Security Implications . . . 69
5.8 Discussion . . . 70

6 A RISC-V Privileged Architecture **78**
6.1 Privileged Software Interfaces . . . 79
6.2 Four Levels of Privilege . . . 80
6.3 A Unified Control Register Scheme . . . 81
6.4 Supervisor Mode . . . 82
6.5 Hypervisor Mode . . . 86
6.6 Machine Mode . . . 86
6.7 Discussion . . . 87

7 Future Directions **88**

Bibliography **90**

A User-Level ISA Encoding **99**

List of Figures

3.1 RV32I user-visible architectural state. 17
3.2 RV32I instruction formats. 18
3.3 Code fragments to load a variable 0x1234 bytes away from the `pc`, with and without the AUIPC instruction. 21
3.4 Sample code for reading the 64-bit cycle counter in RV32. 27
3.5 RV32I user-visible architectural state. 28
3.6 RV64I user-visible architectural state. 29

4.1 Compare-and-swap implemented using load-reserved and store-conditional. . . . 35
4.2 Atomic addition of bytes, implemented with a word-sized LR/SC sequence. . . . 36
4.3 Orderings between accesses mandated by release consistency. The origin of an arrow cannot be perceived to have occurred before the destination of the arrow. 37
4.4 RVF user-visible architectural state. 39
4.5 A routine that computes $\lfloor \log_2 |x| \rfloor$ by extracting the exponent from a floating-point number, with and without the FMV.X.S instruction. 44
4.6 Fused multiply-add instruction format, R4. 45

5.1 Frequency of integer register usage in static code in the SPEC CPU2006 benchmark suite. Registers are sorted by function in the standard RISC-V calling convention. Several registers have special purposes in the ABI: `x0` is hard-wired to the constant zero; `ra` is the link register to which functions return; `sp` is the stack pointer; `gp` points to global data; and `tp` points to thread-local data. The `a`-registers are caller-saved registers used to pass parameters and return results. The `t`-registers are caller-saved temporaries. The `s`-registers are callee-saved and preserve their contents across function calls. 52
5.2 Cumulative frequency of integer register usage in the SPEC CPU2006 benchmark suite, sorted in descending order of frequency. 52
5.3 Frequency of floating-point register usage in static code in the SPEC CPU2006 benchmark suite. Registers are sorted by function in the standard RISC-V calling convention. Like the integer registers, the `a`-registers are used to pass parameters and return results; the `t`-registers are caller-saved temporaries; and the `s`-registers are callee-saved. 53

5.4 Cumulative frequency of floating-point register usage in the SPEC CPU2006 benchmark suite, sorted in descending order of frequency. 54
5.5 Cumulative distribution of immediate operand widths in the SPEC CPU2006 benchmark suite when compiled for RISC-V. Since RISC-V has 12-bit immediates, the immediates in SPEC wider than 12 bits are loaded with multiple instructions and manifest in this data as multiple smaller immediates. 54
5.6 Cumulative distribution of branch offset widths in the SPEC CPU2006 benchmark suite. Branches in the base ISA have 12-bit two's-complement offsets in increments of two bytes ($\pm 2^{10}$ instructions). Jumps have 20-bit offsets ($\pm 2^{18}$ instructions). 55
5.7 Static compression of RVC code compared to RISC-V code in the SPEC CPU2006 benchmark suite, Dhrystone, CoreMark, and the Linux kernel. The SPECfp outlier, `lbm`, is briefly discussed in the next section. 60
5.8 SPEC CPU2006 code size for several ISAs, normalized to RV32C for the 32-bit ISAs and RV64C for the 64-bit ones. Error bars represent ± 1 standard deviation in normalized code size across the 29 benchmarks. 62
5.9 Dynamic compression of RVC code compared to RISC-V code in the SPEC CPU2006 benchmark suite, Dhrystone, CoreMark, and the Linux kernel. 63
5.10 Code snippet from `libquantum`, before and after adjusting the C compiler's cost model to favor RVC registers in hot code. The compiler tweak reduced the size of the code from 30 to 24 bytes. 64
5.11 Speedup of larger caches, associative caches, and RVC over a direct-mapped cache baseline, for a range of instruction cache sizes. 65
5.12 Naïve method to compute the factorial of an integer, both without and with prologue and epilogue millicode calls. 67
5.13 Sample implementations of prologue and epilogue millicode routines for saving and restoring `ra` and `s0`. 67
5.14 Impact on static code size and dynamic instruction count of compressed function prologue and epilogue millicode routines. 68
5.15 Interpretation of eight bytes of RVC code, depending on whether the code is entered at byte 0 or at byte 2. 69

6.1 The same ABI can be implemented by many different privileged software stacks. For systems running on real RISC-V hardware, a hardware abstraction layer underpins the most privileged execution environment. 79

List of Tables

2.1 Summary of several ISAs' support for desirable architectural features. 14

3.1 RV32I opcode map. 18
3.2 Listing of RV32I computational instructions. 19
3.3 Listing of RV32I memory access instructions. 22
3.4 Listing of RV32I control transfer instructions. 23
3.5 Listing of RV32I system instructions. 26
3.6 Listing of RV32I control and status registers. 26
3.7 Listing of additional RV64I computational instructions. 29
3.8 Listing of additional RV128I computational instructions. 31

4.1 Listing of RV32M and (below the line) RV64M instructions. 33
4.2 Listing of RVA instructions. The instructions with the `w` suffix operate on 32-bit words; those with the `d` suffix are RV64A-only instructions that operate on 64-bit words. 37
4.3 Supported rounding modes and their encoding. 40
4.4 Default single-precision NaN for several ISAs. QNaN polarity refers to whether the most significant bit of the significand indicates that the NaN is quiet when set, or quiet when clear. The values come from [87, 67, 54, 47, 3, 8]. 41
4.5 Listing of RVF instructions. 43
4.6 Classes into which the FCLASS instruction categorizes Format of result of FCLASS instruction. 45
4.7 Listing of RVD instructions. 47

5.1 Twenty most common RV64IMAFD instructions, statically and dynamically, in SPEC CPU2006. ADDI's outsized popularity is due not only to its frequent use in updating induction variables but also to its two idiomatic uses: synthesizing constants and copying registers. 56
5.2 Major RVC instruction formats, from [101]. 57
5.3 RV32C and RV64C instruction listing. 72
5.4 RV64C instruction encoding. 73
5.5 Reserved encodings in RV64C. 74

5.6 SPEC CPU2006 code size for several ISAs, normalized to RV32C for the 32-bit ISAs and RV64C for the 64-bit ones. Thm2 is short for ARM Thumb-2; μM is short for microMIPS. 75
5.7 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. 76
5.8 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. 77

6.1 RPA privilege modes and supported privilege mode combinations. 80
6.2 RPA privilege modes and supported privilege mode combinations. 81
6.3 Listing of supervisor-mode control and status registers. 82
6.4 Page table entry types. 84

Acknowledgments

This thesis and the research it represents would not have been possible without the technical, professional, and moral support of many humans. Foremost among them are my advisors, Krste Asanović and Dave Patterson, whom I thank for their service as mentors and as role models, for their relentless encouragement, and for their exceptional patience.

I have the great fortune to have worked with talented colleagues and friends at Berkeley. Special thanks to Yunsup Lee, who co-designed RISC-V; Zhangxi Tan and Sam Williams, who mentored me early in grad school; John Hauser, who taught me more than I thought I wanted to know about computer arithmetic; and Rimas Avižienis, Henry Cook, Chen Sun, and Brian Zimmer, who helped design and fabricate the early RISC-V prototypes. Many thanks to the rest of my research group, who have all been sources of support and inspiration in their own ways.

Thanks to Jonathan Bachrach for spearheading the Chisel HDL effort and for agreeing to be on my quals committee, and to Per-Olof Persson for responding to a cold call to be on my dissertation committee. On a sad note, committee member David Wessel passed away, the loss of a mentor and a friend.

Thanks to John Board and Dan Sorin at Duke, devoted educators who engendered my interest in digital systems and computer architecture. Without a gentle nudge from Dan, I probably wouldn't even have applied to grad school.

Thanks to the unsung heroes of the Par Lab and ASPIRE Lab: the system administrators, Jon Kuroda and Kostadin Ilov, and the administrative staff, Roxana Infante and Tamille Johnson. All of them have gone above and beyond the call of duty.

Much love to my family, who even from from 2,000 miles away have been enormously supportive.

Finally, I am eternally indebted to my friends in Berkeley, who have made my stay here so rewarding. In particular, my roommates—Eric, Nick, Erin, Nick, Jack, David; the cats, Sterling and Barry; and the dog, Blue—have been a source of sanity in an endeavor that has occasionally been far from sane.

Funding Support

Early development of the RISC-V architecture and its original implementations took place in the Par Lab. After that project declared success in 2013, the RISC-V work continued in the ASPIRE Lab. This research could not have happened without the financial support of both labs' industrial and governmental sponsors:

- **Par Lab:** Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support came from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.

- **ASPIRE Lab**: DARPA PERFECT program, Award HR0011-12-2-0016. DARPA POEM program Award HR0011-11-C-0100. The Center for Future Architectures Research (C-FAR), a STARnet center funded by the Semiconductor Research Corporation. Additional support from ASPIRE industrial sponsor, Intel, and ASPIRE affiliates, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

Chapter 1

Introduction

The rapid clip of innovation in computer systems design owes in no small part to the careful design of interfaces between subsystems. Interfaces serve as abstraction layers, enabling researchers to experiment with the design of one component without interfering with the functionality of another. Arguably, the most important abstraction layer in a computer system is the hardware/software interface, an observation that, surprisingly, was not made until the introduction of the IBM 360 [4], 15 years after the introduction of the stored-program computer [105]. IBM announced a line of six computers of vastly different cost and performance that all executed the same software, introducing the concept of the instruction set architecture (ISA) as an entity distinct from its hardware implementation. It is no coincidence that the IBM 360 has long outlived its predecessors.

More recently, *open* computing standards, like Ethernet [43] and floating-point arithmetic [7], have proved wildly successful, allowing free-market competition on technical merit while supporting compatible interchange of data and interconnection of systems. It is thus remarkable that all of today's popular ISAs are proprietary standards. Of course, it is natural that the stewards of these ISAs seek to protect their intellectual property, but keeping the standards closed stymies innovation and artificially inflates the cost of microprocessors [17]. Yet, there is no good technical reason for this state of affairs.

We seek to upend the status quo. This thesis describes the design of the RISC-V instruction set architecture, a completely free and open ISA. Leveraging three decades of hindsight, RISC-V builds and improves on the original Reduced Instruction Set Computer (RISC) architectures. The result is a clean, simple, and modular ISA that is well suited to low-power embedded systems and high-performance computers alike.

Our ambitions were not always so grand. Yunsup Lee, Krste Asanović, David Patterson, and I conceived RISC-V in the summer of 2010 as an ISA for research and education at Berkeley. Two of our research ventures, RAMP Gold [92] and Maven [60], had just wound down. These projects were based around the SPARC ISA and a lightly modified MIPS ISA, respectively, and we sought to unify behind a single architecture for the next round of projects. For reasons discussed in Chapter 2, we found neither SPARC nor MIPS appealing. After weighing our options, we embarked on what we expected would be a semester-long

effort to make a clean-slate design of a new ISA. To say we underestimated the task would be a charitable understatement: we completed the user-level instruction set architecture four years later. The endeavor proved to be much deeper than an engineering task. Reexploring ISA design issues raised interesting questions and, in the end, resulted in an architecture superior to its RISC forebears. Chapters 3 and 4 describe the RISC-V user-level ISA and discuss these architectural design decisions. Chapters 5 and 6 describe two ongoing efforts: an ISA extension for greater code density and a privileged architecture specification.

Before designing RISC-V, we carefully considered the possibility of adopting an existing ISA. The next chapter explains why we ultimately chose not to do so.

Chapter 2

Why Develop a New Instruction Set?

Perhaps the most common question we have been asked over the course of designing RISC-V is why there is any need for a new instruction set architecture (ISA). After all, there are several commercial ISAs in popular use, and reusing one of them would avoid the significant effort and cost of porting software to a new one. In our minds, two main downsides outweighed that consideration.

First, all of the popular commercial ISAs are proprietary. Their vendors have a lucrative business selling implementations of their ISAs, be it in the form of IP cores or silicon, and so they do not welcome freely available implementations that might erode their profits. While this consideration does not itself prohibit all forms of academic computer architecture research using these ISAs, it does preclude the creation and sharing of full RTL implementations of them. It also erects a barrier to the commercialization of successful research ideas.

Of equal importance is the massive complexity of the popular commercial instruction sets. They are quite difficult to fully implement in hardware, and yet there is little incentive to create simpler subset ISAs: without a complete implementation, unmodified software cannot run, undermining the justification for using an existing ISA. Furthermore, while some degree of complexity is necessary, or at least beneficial, these instruction sets tend not to be complicated for sound technical reasons. Much simpler instruction sets can lead to similarly performant systems.

Even so, we carefully considered the possibility of adopting an existing instruction set, rather than developing our own. In this chapter, we discuss several ISAs we considered and why we ultimately rejected them.

2.1 MIPS

The MIPS instruction set architecture is a quintessential RISC ISA. Originally developed at Stanford in the early 1980s [38], its design was heavily influenced by the IBM 801 minicomputer [81]. Both are load-store architectures with general-purpose registers, wherein memory

is only accessed by instructions that copy data to and from registers, and arithmetic only operates on the registers. This design reduces the complexity of both the instruction set and the hardware, facilitating inexpensive pipelined implementations while relying on improved compiler technology. The MIPS was first commercially implemented in the R2000 processor in 1986 [53].

In its original incarnation, the MIPS user-level integer instruction set comprised just 58 instructions and was straightforward to implement as a single-issue, in-order pipeline. Over 30 years, it has evolved into a much larger ISA, now with about 400 instructions [63]. While simple microarchitectural realizations of MIPS-I are well within the grasp of academic computer architects, the ISA has several technical drawbacks that make it less attractive for high-performance implementations:

- The ISA is over-optimized for a specific microarchitectural pattern, the five-stage, single-issue, in-order pipeline. Branches and jumps are delayed by one instruction, complicating superscalar and superpipelined implementations. The delayed branches increase code size and waste instruction issue bandwidth when the delay slot cannot be suitably filled. Even for the classic five-stage pipeline, dropping the delay slot and adding a small branch target buffer typically results in better absolute performance and performance per unit area.

  Other pipeline hazards, including data hazards on loads, multiplications, and divisions, were exposed in MIPS-I, but later revisions of the ISA removed these warts, reflecting the fact that interlocking on these hazards is both simpler for the software and can offer higher performance. The branch delay slot, on the other hand, cannot be removed while preserving backwards compatibility.

- The ISA provides poor support for position-independent code (PIC), and hence dynamic linking. The direct jump instructions are pseudo-absolute, rather than relative to the program counter, thereby rendering them useless in PIC; instead, MIPS uses indirect jumps exclusively, at a significant code size and performance cost. (The 2014 revision of MIPS has improved PC-relative addressing, but PC-relative function calls still generally take more than one instruction.)

- Sixteen-bit-wide immediates consume substantial encoding space, leaving only a small fraction of the opcode space available for ISA extensions—about $\frac{1}{64}$ as of the 2014 revision. When the MIPS architects sought to reduce code size with a compressed instruction encoding, they had no choice but to create a second instruction encoding, enabled with a mode switch, because they could not fit the new instructions into the original encoding.

- Multiplication and division use special architectural registers, increasing context size, instruction count, code size, and microarchitectural complexity.

- The ISA presupposes that the floating-point unit is a separate coprocessor and is suboptimal for single-chip implementations. For example, floating-point to integer conversions write their results to the floating-point register file, typically necessitating an extra move instruction to make use of the result. Exacerbating this cost, moves between the integer and floating-point register files have a software-exposed delay slot.
- In the standard ABI, two of the integer registers are reserved for kernel software, reducing the number of registers available to user programs. Even so, the registers are of limited use to the kernel because they are not protected from user access.
- Handling misaligned loads and stores with special instructions consumes substantial opcode space and complicates all but the simplest implementations.
- The architects omitted integer magnitude compare-and-branch instructions, a clock rate/CPI tradeoff that is less appropriate today with the advent of branch prediction and the move to static CMOS logic.

Technical issues aside, MIPS is unsuitable for many purposes because it is a proprietary instruction set. Historically, MIPS Technologies' patent on the misaligned load and store instructions [37] had prevented others from fully implementing the ISA. In one instance, a lawsuit targeted a company whose MIPS implementations *excluded* the instructions, claiming that emulating the instructions in kernel software still infringed on the patent [98]. While the patent has since expired, MIPS remains a trademark of Imagination Technologies; MIPS compatibility cannot be claimed without their permission.

2.2 SPARC

Oracle's SPARC architecture, originally developed by Sun Microsystems, traces its lineage to the Berkeley RISC-I and RISC-II projects [78, 56]. The most recent 32-bit version of the ISA, SPARC V8 [87], is not unduly complicated: the user-level integer ISA has a simple, regular encoding and comprises just 90 instructions. Hardware support for IEEE 754-1985 floating-point adds another 50 instructions, and the supervisor mode another 20. Nevertheless, several ISA design decisions make it quite a bit less attractive to implement than the MIPS-I:

- To accelerate function calls, SPARC employs a large, windowed register file. At procedure call boundaries, the window shifts, giving the callee the appearance of a fresh register set. This design obviates the need for callee-saved register save and restore code, which reduces code size and typically improves performance. If the procedure call stack's working set exceeds the number of register windows, though, performance suffers dramatically: the operating system must be routinely invoked to handle the window overflows and underflows. The vastly increased architectural state increases

the runtime cost of context switches, and the need to invoke the operating system to flush the windows precludes pure user-level threading altogether.

The register windows come at a significant area and power cost for all implementations. Techniques to mitigate their cost complicate superscalar implementations in particular. For example, to avoid provisioning a large number of ports across the entire architectural register set, the UltraSPARC-III provisions a shadow copy of the active register window [86]. The shadow copy must be updated on register window shifts, causing a pipeline break on most function calls and returns. Fujitsu's out-of-order execution implementations went to similarly heroic lengths [5], folding the register window addressing logic into the register renaming circuitry.

- Branches use condition codes, which add to the architectural state and complicate implementations by creating additional dependences between some instructions. Out-of-order microarchitectures with register renaming need to separately rename the condition codes to obviate a frequent serialization bottleneck. The lack of a fused compare-and-branch instruction also increases static and dynamic instruction count for common code sequences.

- The instructions that load and store adjacent pairs of registers are attractive for simple microarchitectures, since they increase throughput with little additional hardware complexity. Alas, they complicate implementations with register renaming, because the data values are no longer physically adjacent in the register file.

- Moves between the floating-point and integer register files must use the memory system as an intermediary, limiting performance for mixed-format code.

- The ISA exposes imprecise floating-point exceptions by way of an architecturally exposed *deferred-trap queue*, which provides supervisor software with the information to recover the processor state on such an exception.

- The only atomic memory operation is fetch-and-store, which is insufficient to implement many wait-free data structures [40].

SPARC shares many of the myopic ISA features of the other 1980s RISC architectures. It was designed to be implemented in a single-issue, in-order, five-stage pipeline, and the ISA reflects this assumption. SPARC has branch delay slots and myriad exposed data and control hazards, which complicate code generation and are no help to more aggressive implementations. Additionally, support for position-independent data addressing is lacking. Finally, SPARC cannot be readily retrofitted to support a compressed ISA extension, as it lacks sufficient free encoding space[1].

[1] As compared to MIPS, SPARC's architects wisely conserved opcode space by using smaller 13-bit immediates for most instructions. Alas, they squandered it in other ways. The `CALL` instruction supports unconditional control transfers to anywhere in the 32-bit address space with a single instruction. This

Unlike the other commercial RISCs, SPARC V8 is an open standard [44], much to Sun's credit. SPARC International continues to grant permissive licenses of V8 and V9, the 64-bit ISA, for a $99 administrative fee. The open ISA has led to freely available implementations [26, 73], two of which are derivatives of Sun's own Niagara microarchitecture. Alas, continued development of the Oracle SPARC Architecture [74] is proprietary, and high-performance software is likely to follow their lead, leaving behind implementations of the older, open instruction set.

2.3 Alpha

Digital Equipment Corporation's architects had the benefit of several years of hindsight when they defined their RISC ISA, Alpha [3], in the early 1990s. They omitted many of the least attractive features of the first commercial RISC ISAs, including branch delay slots, condition codes, and register windows, and created a 64-bit address-space ISA that was cleanly designed, simple to implement, and capable of high performance. Additionally, the Alpha architects carefully isolated most of the details of the privileged architecture and hardware platform behind an abstract interface, the Privileged Architecture Library (PALcode) [77].

Nevertheless, DEC over-optimized Alpha for in-order microarchitectures and added a handful of features that are less than desirable for modern implementations:

- In the pursuit of high clock frequency, the original version of the ISA eschewed 8- and 16-bit loads and stores, effectively creating a word-addressed memory system. To recoup performance on applications that made extensive use of these operations, they added special misaligned load and store instructions and several integer instructions to speed realignment. The architects eventually realized the error of their ways—application performance still suffered, and it was impossible to implement some device drivers—and added the sub-word loads and stores to the ISA. But they were still saddled with the old alignment-handling instructions, which were no longer terribly useful.

- To facilitate out-of-order completion of long-latency floating-point instructions, Alpha has an imprecise floating-point trap model. This decision might have been acceptable in isolation, but the ISA also defines that the exception flags and default values, if desired, must be provided by software routines. The combination is disastrous for IEEE-754-compliant programs: trap barrier instructions must be inserted after most floating-point arithmetic instructions (or, if a baroque list of code generation restrictions is followed, once per basic block).

simplifies the linking model, but optimizes for the vastly uncommon case at significant cost: `CALL` consumes an entire $\frac{1}{4}$ of the ISA's opcode space.

- Alpha lacks an integer division instruction, instead suggesting the use of a software Newton-Raphson iteration scheme. This approach greatly increases instruction count for some programs and saves only a small amount of hardware. The surprising consequence is that floating-point division is significantly faster than integer division on most implementations.

- As with its predecessor RISCs, no forethought was given to a possible compressed instruction set extension, and so not enough opcode space remains to retrofit one.

- The ISA contains conditional moves, which complicate microarchitectures with register renaming: in the event that the move condition is not met, the instruction must still copy the old value into the new physical destination register. This effectively makes the conditional move the only instruction in the ISA that reads three source operands.

  Indeed, DEC's first implementation with out-of-order execution employed some chicanery to avoid the extra datapath for this instruction. The Alpha 21264 executed the conditional move instruction by splitting it into two micro-operations, the first of which evaluated the move condition and the second of which performed the move [57]. This approach also required that the physical register file be widened by one bit to hold the intermediate result[2].

The Alpha also highlights an important risk of using commercial ISAs: they can die. Not long after Compaq purchased what remained of the faltering DEC in the late 1990s, they chose to phase out the Alpha in favor of Intel's Itanium architecture. Compaq sold the Alpha intellectual property to Intel [80], and soon thereafter, HP, who had since purchased Compaq, produced the final Alpha implementation in 2004 [55].

2.4 ARMv7

ARMv7 is a popular 32-bit RISC-inspired ISA, and by far the most widely implemented architecture in the world [10]. As we weighed whether or not to design our own instruction set, ARMv7 was a natural alternative due to the great quantity of software that has been ported to the ISA and to its ubiquity in embedded and mobile devices. Ultimately, we could not adopt ARMv7 because it is a closed standard. Subsetting the ISA or extending it with new instructions is explicitly disallowed; even microarchitectural innovation is restricted to those who can afford what ARM refers to as an architectural license.

Had intellectual property encumbrances not been an issue, though, there are several technical deficiencies in ARMv7 that strongly disinclined us to use it:

[2]By contrast, the out-of-order MIPS R10000 processor implemented conditional moves by provisioning a one-bit-wide register file that summarized the zeroness of each physical register. This approach is simpler than the Alpha 21264's, but it requires a third wakeup port in the issue window, increasing power and area and possibly cycle time.

- At the time, there was no support for 64-bit addresses, and the ISA lacked hardware support for the IEEE 754-2008 standard. (ARMv8 rectified these deficiencies, as discussed in the next section.)

- The details of the privileged architecture seep into the definition of the user-level architecture. This concern is not merely aesthetic. ARMv7 is not *classically virtualizable* [32] because, among other reasons, the return-from-exception instruction, `RFE`, is not defined to trap when executed in user mode [79, 8]. ARM has added a hypervisor privilege mode in recent revisions of the architecture, but as of this writing, it remains impossible to classically virtualize without dynamic binary translation.

- ARMv7 is packaged with a compressed ISA with fixed-width 16-bit instructions, called Thumb. Thumb offers competitive code size but low performance, especially on floating-point-intensive code. A variable-length instruction set, Thumb-2, followed later, providing much higher performance. Unfortunately, since Thumb-2 was conceived after the base ARMv7 ISA was defined, the 32-bit instructions in Thumb-2 are encoded differently than the 32-bit instructions in the base ISA. (The 16-bit instructions in Thumb-2 are also encoded differently than the 16-bit instructions in the original Thumb ISA.) Effectively, the instruction decoders need to understand three ISAs, adding to energy, latency, and design cost.

- The ISA has many features that complicate implementations. It is not a truly general-purpose register architecture: the program counter is one of the addressable registers, meaning that nearly any instruction can change the flow of control. Worse yet, the least-significant bit of the program counter reflects which ISA is currently executing (ARM or Thumb)—the humble ADD instruction can change which ISA is currently executing on the processor! The use of condition codes for branches and predication further complicates high-performance implementations.

ARMv7 is vast and complicated. Between ARM and Thumb, there are over 600 instructions in the integer ISA alone[3]. NEON, the integer SIMD and floating-point extension, adds hundreds more. Even if it had been legally feasible for us to implement ARMv7, it would have been quite challenging technically.

2.5 ARMv8

In 2011, a year after we started the RISC-V project, ARM announced a completely redesigned ISA, ARMv8, with 64-bit addresses and an expanded integer register set. The new architecture removed several features of ARMv7 that complicated implementations: for example, the program counter is no longer part of the integer register set; instructions are no

[3]This counts all user-level instructions in ARMv7-A, excluding NEON. Instructions that set the condition codes are considered distinct from those that do not. Different addressing modes also count as distinct.

longer predicated; the load-multiple and store-multiple instructions were removed; and the instruction encoding was regularized. But many warts remain, including the use of condition codes and not-quite-general-purpose registers (the link register is implicit and, depending on the context, `x31` is either the stack pointer or is hard-wired to zero). And more blemishes were added still, including a massive subword-SIMD architecture that is effectively mandatory[4]. Overall, the ISA is complex and unwieldy: there are 1070 instructions, comprising 53 formats and and eight data addressing modes [18], all of which takes 5,778 pages to document [9]. Given that, it is perhaps surprising that important features were left out: for example, the ISA lacks a fused compare-and-branch instruction.

Like most ISAs we considered, ARMv8 closely intermingles the user and privileged architectures, often in ways that expose the underlying implementation. In one inexplicable example—which combines complicated semantics, undefined behavior, and register-dependent properties of allegedly general-purpose registers—the load-pair instruction may expose to user-space an imprecise exception:

> "If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and (t == n || t2 == n) && n != 31, then one of the following behaviors can occur:
>
> - The instruction is UNDEFINED.
> - The instruction executes as a NOP.
> - The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated." [9]

Additionally, with the introduction of ARMv8, ARM has dropped support for a compressed instruction encoding. The compact Thumb instruction set has not been brought along to the 64-bit address space. It is true that ARMv8 is quite compact for an ISA with fixed-width instructions, but, as we show in Chapter 5, it cannot compete in code size with a variable-length ISA. In what is surely not a coincidence, ARM's first 64-bit implementations have 50% larger instruction caches than their 32-bit counterparts [13, 14].

Finally, like its predecessor, ARMv8 is a closed standard. It cannot be subsetted, making implementations far too bulky to serve as embedded processors or as control units for custom accelerators. In fact, tightly coupled coprocessors are essentially impossible to design around this instruction set, since it cannot be extended by anyone but ARM. Even architects content to innovate in the microarchitecture cannot do so without a costly license, greatly limiting the number of people who can implement ARMv8.

[4]Presumably in a gambit to prevent the ISA fragmentation that plagued its earlier ISAs, ARM requires that implementations of ARMv8 that run general-purpose operating systems implement the entire ISA, including the Advanced SIMD instructions. For these systems, there is no software floating-point ABI.

2.6 OpenRISC

The OpenRISC project is an open-source processor design effort that evolved out of the educational DLX architecture of Hennessy and Patterson's influential computer architecture textbook [39]. As a free and open ISA, OpenRISC is legally suitable for use in academic, research, and industrial implementations. Like the DLX, though, it has several technical drawbacks that limit its applicability:

- The OpenRISC project is principally an open processor design, rather than an open ISA specification. The ISA and implementation are very tightly coupled.
- The fixed 32-bit encoding with 16-bit immediates precludes a compressed ISA extension.
- The 2008 revision of the IEEE 754 standard is not supported in hardware.
- Condition codes, used for branches and conditional moves, complicate high-performance implementations.
- The ISA provides poor support for position-independent data addressing.
- OpenRISC is not classically virtualizable because the return-from-exception instruction, `L.RFE`, is defined to function normally in user mode, rather than trapping[5] [72].

When we first investigated OpenRISC in 2010, the ISA had two additional drawbacks: mandatory branch delay slots, and no 64-bit address space variant. To the architects' credit, both of these have been rectified: the delay slots have become optional, and the 64-bit version has been defined (but, to our knowledge, never implemented). Ultimately, we thought it was best for our purposes to start from a clean slate, rather than modifying OpenRISC accordingly.

2.7 80x86

Intel's 8086 architecture has, over the course of the last four decades, become the most popular instruction set in the laptop, desktop, and server markets. Outside of the domain of embedded systems, virtually all popular software has been ported to, or was developed for, the x86. The reasons for its popularity are myriad: the architecture's serendipitous availability at the inception of the IBM PC; Intel's laser focus on binary compatibility; their aggressive microarchitectural implementations; and their leading-edge fabrication technology.

The design quality of the instruction set architecture is not one of them.

[5] In addition to the virtualization hole, `L.RFE`'s behavior is also a potential vector for a side channel attack, since it enables user code to determine the PC at which the most recent interrupt occurred.

In 1994, AMD's 80x86 architect, Mike Johnson, famously quipped, "The x86 really isn't all that complex—it just doesn't make a lot of sense" [85]. At the time, the comment humorously understated the ISA's historical baggage. Over the course of the last two decades, it has proved surprisingly inaccurate: the x86 of 2015 is extremely complex. It now comprises 1300 instructions, myriad addressing modes, dozens of special-purpose registers, and multiple address-translation schemes. It should come as no surprise that, following the lead of AMD's K5 microarchitecture [85], all of Intel's out-of-order execution engines dynamically translate x86 instructions into an internal format that more closely resembles a RISC-style instruction set.

The x86's complexity would be justifiable if it ultimately resulted in more efficient processors. Alas, the second half of Mr. Johnson's barb rings true today. One wonders how much thought went into the design:

- The ISA is not classically virtualizable, since some privileged instructions silently fail in user mode rather than trapping. VMware's engineers famously worked around this deficiency with intricate dynamic binary translation software [23].

- The ISA has instruction lengths of any integer number of bytes up to 15, but the less-numerous short opcodes have been used capriciously. For example, in IA-32, Intel's 32-bit incarnation of the 80x86, six of the 256 8-bit opcodes accelerate the manipulation of binary-coded decimal numbers—operations so esoteric that the GNU compiler does not even emit these instructions. (Although x86-64 dropped this particularly egregious example, numerous wasteful uses of the 8-bit opcode space remain, including an instruction to check for pending floating-point exceptions in the deprecated x87 floating-point unit.)

- The ISA has an anemic register set. The 32-bit architecture, IA-32, has just eight integer registers. Spills to the stack are so common that, to reduce pipeline occupancy and data cache traffic, recent Intel microarchitectures have a special functional unit that manages the stack pointer's value and caches the top several words of the stack [46].

  Recognizing this deficiency, AMD's 64-bit extension, x86-64, doubled the number of integer registers to 16. Even so, many programs—particularly those that would benefit from compiler optimizations like loop unrolling and software pipelining—still face register pressure.

- Exacerbating the paucity of architectural registers, most of the integer registers perform special functions in the ISA. For example, integer dividends are implicitly sourced from the DX and AX register pair. Shift amounts only come from the CX register, which also serves as the induction variable register for string operations. ESI provides the address for the post-increment load addressing mode, whereas EDI does so for post-increment stores. In general, this design pattern results in inefficient shuffling of data between registers and the stack.

- Worse still, most x86 instructions have only a destructive form that overwrites one of the source operands with the result. Frequently, this necessitates extra moves to preserve values that remain live across destructive instructions.

- Several ISA features, including implicit condition codes and predicated moves, are onerous to implement in aggressive microarchitectures. Yet, their complexity often does not result in higher performance because their semantics were ill-conceived. For example, x86 provides a conditional load instruction, but, if the unconditional load were to cause an exception, it is implementation-defined whether the conditional version would do so. Thus, a compiler can only rarely use this instruction to perform the if-conversion optimization.

  Recognizing the inefficiency of their conditional operations, Intel's recent implementations go to some lengths to fuse comparison instructions and branch instructions into internal compare-and-branch operations.

These ISA decisions have profound effects on static code size. As we show in Chapter 5, what could otherwise be a very dense instruction encoding is not at all: IA-32 is only marginally denser than the fixed-width 32-bit ARMv7 encoding, and x86-64 is quite a bit less dense than ARMv8.

Despite all of these flaws, x86 typically encodes programs in fewer dynamic instructions than RISC architectures, because the x86 instructions can encode multiple primitive operations. For example, the C expression `x[2] += 13` might compile in MIPS to the three-instruction sequence `lw r5, 8(r4); addiu r5, r5, 13; sw r5, 8(r4)`, but the single instruction `addl 13, 8(eax)` suffices in IA-32. This dynamic instruction density has some advantages: for example, it can reduce the instruction fetch energy cost. But it complicates implementations of all stripes. In this example, a regular pipeline would exhibit two structural hazards, since the instruction performs two memory accesses and two additions[6].

Finally, the 80x86 is a proprietary instruction set. Architects brave enough to attempt to implement an x86 microprocessor competitive with Intel's offerings are likely to face legal hurdles: Intel has historically been quite litigious, even in the face of their own antitrust troubles [93].

2.8 Summary

Table 2.1 summarizes these instruction sets' support for several features we consider essential for a modern general-purpose ISA. All of the architectures lack at least two important technical features. ARMv8, which comes closest, is a proprietary standard. The two open ISAs,

[6]Some x86 implementations, like Intel's in-order 486 [51] and Pentium [24] pipelines and Cyrix's out-of-order M1 [35], handled these structural hazards with hard-wired control. More recent ones, starting with the AMD K5 [85] and Intel Pentium Pro [36], resolve the hazard by breaking these instructions into a sequence of simpler micro-operations.

	MIPS	SPARC	Alpha	ARMv7	ARMv8	OpenRISC	80x86
Free and Open		✓				✓	
64-bit Addresses	✓	✓	✓		✓	✓	✓
Compressed Instructions	✓			✓			Partial
Separate Privileged ISA			✓				
Position-Indep. Code	Partial			✓	✓		✓
IEEE 754-2008					✓		✓
Classically Virtualizable	✓	✓	✓		✓		

Table 2.1: Summary of several ISAs' support for desirable architectural features.

SPARC and OpenRISC, lack several crucial architectural features. All of the ISAs, except, arguably, the DEC Alpha, have other properties that substantially increase implementation complexity, especially for high-performance implementations.

Given these limitations, we saw fit to develop our own instruction set. With the benefit of hindsight, we created RISC-V, a free and open ISA that avoids these technical pitfalls and is straightforward to implement in many microarchitectural styles. Its design is the subject of the next chapter.

Chapter 3

The RISC-V Base Instruction Set Architecture

After surveying the contemporary ISA landscape and deeming the existing options unsuitable for our research and educational purposes, we set out to define our own ISA. Building on the legacy of the RISC-I [78], RISC-II [56], SOAR [83], and SPUR [42] projects, ours was the fifth major RISC ISA design effort at UC Berkeley, and so we named it RISC-V. As one of our goals in defining RISC-V was to support research in data-parallel architectures, the Roman numeral 'V' also conveniently served as an acronymic pun for "Vector."

The guiding principle in defining RISC-V was to make an ISA suitable for nearly any computing device. This goal has two direct consequences. First, RISC-V should not be over-architected for any particular microarchitectural pattern, implementation fabric, or deployment target. The architects of many of the ISAs discussed in Chapter 2 made decisions that over-optimized for the originally intended style of implementation (e.g., MIPS' delayed branches and SPARC's condition codes). Alas, different application domains demand different microarchitectural styles, and such features complicate some of those implementations. Similarly, not all domains demand all of the features of a rich ISA (e.g., ARMv8's SIMD); to provision them anyway would increase cost and reduce efficiency. A running theme in the design decisions described in this chapter is avoiding architectural techniques that would provide minor benefit to some RISC-V implementations at undue expense to others.

The second, more important consequence of our goal to make RISC-V ubiquitous is that the ISA must be open and free to implement. The benefits of an open standard are multifold, but perhaps the most important one is the potential abundance of processor implementations. Free, open-source implementations will reduce the cost of building new systems. Free-market competition between open and proprietary implementations alike should ultimately spur microarchitectural innovation. Concerns about the viability of intellectual property providers are assuaged, since open-source implementations always provide a second source. The barrier to academic-industrial interactions is lowered if the ivory tower and the corporate world share common standards and implementations. Finally, an open standard ameliorates one set of security concerns: entities that do not trust certain implementations of the standard—

perhaps due to fears of industrial espionage, or of meddlesome governments—can instead devise their own.

Concordant with making RISC-V broadly applicable, we had several specific technical goals in defining the ISA. We sought to:

- *Separate the ISA into a small base ISA and optional extensions.* The base ISA is lean enough to be suitable for educational purposes and for many embedded processors, including the control units of custom accelerators, yet it is complete enough to run a modern software stack. The extensions improve performance for computational workloads and provide support for multiprocessing.

- *Support both 32-bit and 64-bit address spaces*, as we predict the former will continue to be popular in small systems for centuries while the latter is desirable even for modest personal computers. We have also defined a 128-bit address space variant, which we discuss in Section 3.4.

- *Facilitate custom ISA extensions*, including tightly coupled functional units and loosely coupled coprocessors.

- *Support variable-length instruction set extensions*, both for improved code density and for expanding the space of possible custom ISA extensions.

- *Provide efficient hardware support for modern standards*, including the IEEE-754 2008 floating-point standard [7] and the C11 and C++11 programming languages [48, 49].

- *Orthogonalize the user ISA and privileged architecture*, allowing full virtualizability and enabling experimentation in the privileged ISA while maintaining user application binary interface (ABI) compatibility.

We believe that we have met these goals. In the remainder of this chapter, we describe the design of the RISC-V base instruction set architecture; the standard extensions are the subject of Chapter 4.

The three base ISAs—RV32I, RV32E, and RV64I—are distinct entities, but their designs are very closely related. RV32I and RV64I differ primarily in the width of the registers and the size of the memory address space. RV32E is a variant of RV32I with fewer registers, meant for deeply embedded systems where every transistor counts.

3.1 The RV32I Base ISA

RV32I is the base 32-bit integer ISA. It is a simple instruction set, comprising just 47 instructions, yet it is complete enough to form a compiler target and satisfy the basic requirements of modern operating systems and runtimes. Eight of the instructions are system instructions (system calls and performance counters) that can be implemented as a single trapping

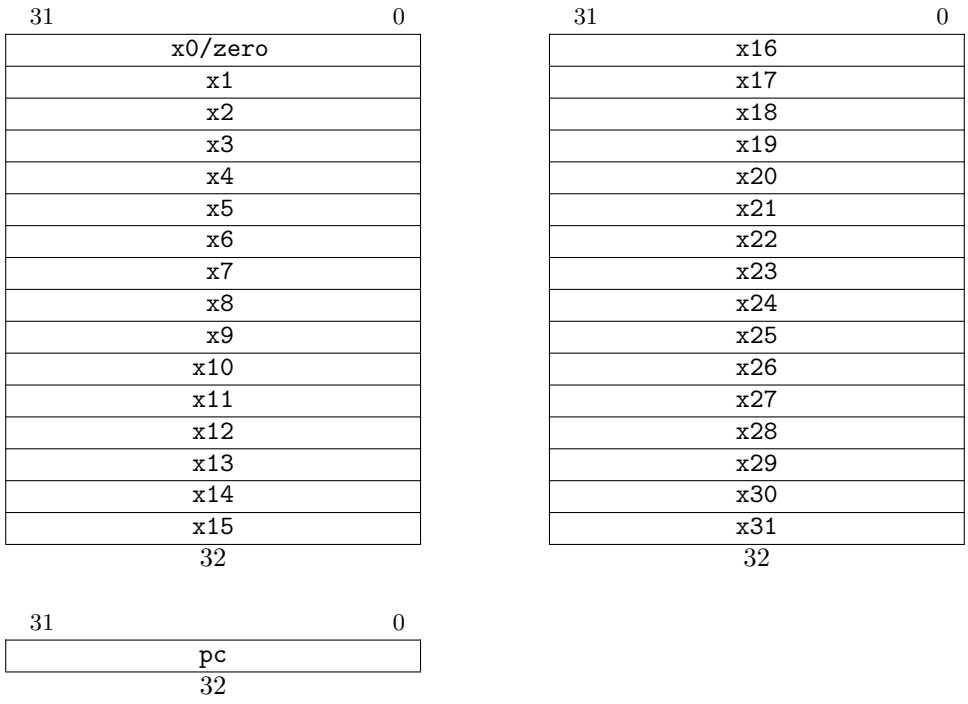

Figure 3.1: RV32I user-visible architectural state.

instruction, reducing the number of mandatory user-level hardware instructions to 40. As with many RISC instruction sets, the remaining instructions fall into three categories: computation, control flow, and memory access. RISC-V is a load-store architecture, in which arithmetic instructions operate only on the registers, and only loads and stores transfer data to and from memory.

There are 31 general-purpose integer registers in RV32I, named `x1`–`x31`, each 32 bits wide. (The register specifier `x0` names the constant zero; it can also be used as a destination register to discard an instruction's result.) The only additional register is the program counter, `pc`, which holds the byte address of the current instruction. As Figure 3.1 shows, the entirety of the user-visible architectural state totals 1024 bits.

Instructions in RV32I are 32 bits long and must be stored naturally aligned in memory, in little-endian byte order[1]. Six instruction formats, which Figure 3.2 depicts, comprise the 47 instructions: four major formats, R, I, S, and U; and two variants, SB and UJ, which are identical to S and U except for the immediate operand encoding. Instructions in these

[1]The choice of memory system endianness is somewhat arbitrary. Some computations, such as IP packet processing and C string manipulation, favor big-endianness. We chose little-endianness because it is currently dominant in general-purpose computing: x86 is little-endian, and, while ARM is bi-endian, little-endian software is more common. While portable software should never rely on memory system endianness, much software, in practice, does. Adopting the most popular choice reduces the effort to port low-level software to RISC-V.

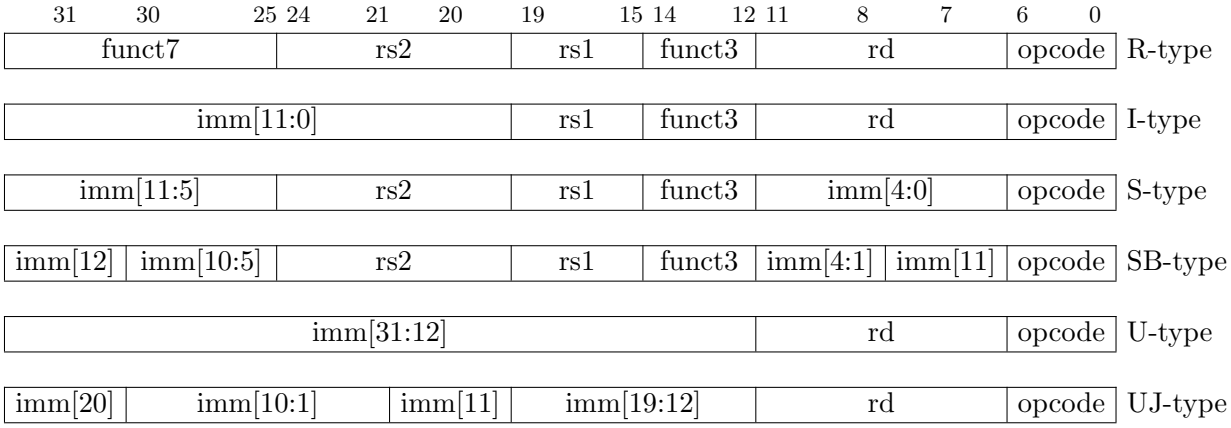

Figure 3.2: RV32I instruction formats.

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111
00	Loads	*F Ext.*		Fences	Arithmetic	AUIPC	*RV64I*	
01	Stores	*F Ext.*		*A Ext.*	Arithmetic	LUI	*RV64I*	
10	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*		*RV128I*	
11	Branches	JALR		JAL	System		*RV128I*	

Table 3.1: RV32I opcode map.

formats source up to two register operands, identified by *rs1* and *rs2*, and produce up to one register result, identified by *rd*. An important feature of this encoding is that these register specifiers, when present, always occupy the same position in the instruction. This property allows register fetch to proceed in parallel with instruction decoding, ameliorating a critical path in many implementations.

Another feature of this encoding scheme is that generating the immediate operand from the instruction word is inexpensive. Of the 32 bits in the immediate operand, seven always come from the same position in the instruction, including the sign bit, which, due to its high fan-out, is the most critical. 24 more bits come from one of two positions, and the final immediate bit has three sources. The SB and UJ formats, which have their immediates scaled by a factor of two, rotate the bits in the immediate, rather than using hardware muxes to do so, as was the case in MIPS, SPARC, and Alpha. This design reduces hardware cost for low-end implementations that reuse the ALU datapath to compute branch targets.

Table 3.1 depicts RV32I's major opcode allocation. Major opcodes are seven bits wide, but in the base ISAs, the two least-significant bits are set to `11`. We reserve the remaining $\frac{3}{4}$ of the encoding space for an ISA extension that significantly improves code density, which is the subject of Chapter 5. RV32I consumes 11 of the 32 major opcodes that remain. The other base ISAs use another four major opcodes, and the standard extensions, the subject

Instruction		Format	Meaning
`add`	`rd, rs1, rs2`	R	Add registers
`sub`	`rd, rs1, rs2`	R	Subtract registers
`sll`	`rd, rs1, rs2`	R	Shift left logical by register
`srl`	`rd, rs1, rs2`	R	Shift right logical by register
`sra`	`rd, rs1, rs2`	R	Shift right arithmetic by register
`and`	`rd, rs1, rs2`	R	Bitwise AND with register
`or`	`rd, rs1, rs2`	R	Bitwise OR with register
`xor`	`rd, rs1, rs2`	R	Bitwise XOR with register
`slt`	`rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu`	`rd, rs1, rs2`	R	Set if less than register, unsigned
`addi`	`rd, rs1, imm[11:0]`	I	Add immediate
`slli`	`rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli`	`rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai`	`rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi`	`rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori`	`rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori`	`rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti`	`rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu`	`rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui`	`rd, imm[31:12]`	U	Load upper immediate
`auipc`	`rd, imm[31:12]`	U	Add upper immediate to `pc`

Table 3.2: Listing of RV32I computational instructions.

of Chapter 4, use eight. Nine major opcodes remain available for ISA extensions. In [102], we describe in detail how we intend to apportion them, but at least two major opcodes will remain reserved for nonstandard extensions.

Appendix A shows the encoding of all RV32I instructions.

Computational Instructions

RV32I comprises 21 computational instructions, including arithmetic, logic, and comparisons. These instructions, which Table 3.2 summarizes, operate on the integer registers; some of the instructions take an additional immediate operand. The computational instructions operate on both signed and unsigned integers. Signed integers use the two's complement representation. All immediate operands are sign-extended, even in contexts where the immediate represents an unsigned quantity. This property reduces the descriptive complexity of the ISA, and actually results in better performance in some cases[2].

[2]MIPS zero-extended some immediates, such as for the bitwise logical operations. This choice requires an extra instruction for operations like masking off the least-significant bit of a register.

The arithmetic operations are addition, subtraction, and bitwise shifts. The R-type instructions ADD and SUB perform addition and subtraction, respectively, on the values in registers *rs1* and *rs2*, writing the result to register *rd*. SLL, SRL, and SRA shift the value in *rs1* by the least-significant five bits of register *rs2*, performing logical left, logical right, and arithmetic right shifts, respectively[3]. I-type instructions ADDI, SLLI, SRLI, and SRAI perform the same operation as their counterparts lacking the letter I, but the right-hand operand comes from the 12-bit signed immediate instead of register *rs2*[4].

The logical operations perform bitwise Boolean operations. AND, OR, and XOR perform their eponymous operations on the values in registers *rs1* and *rs2*, then write the result to register *rd*. ANDI, ORI, and XORI do the same, but they substitute the 12-bit immediate in *rs2*'s stead. Since the immediate operand is sign-extended, RISC-V implements bitwise logical inversion, a.k.a. NOT, using XORI with an immediate of -1. In contrast, MIPS, whose bitwise operations zero-extend their immediates, had to provide an additional instruction, NOR, for this purpose. NOR is otherwise rarely used.

The comparison operations perform arithmetic magnitude comparisons. SLT and SLTU perform signed and unsigned less-than comparisons between *rs1* and *rs2*, writing the Boolean result (0 or 1) to register *rd*. Their I-type counterparts SLTI and SLTIU do the same, but source the right-hand side of the comparison from the 12-bit sign-extended immediate instead of *rs2*. These instructions also provide two common idioms. SLTIU with an immediate of 1 computes whether *rs1* is equal to zero; we call this pseudo-instruction SEQZ. SLTU with *rs1*=`x0` computes whether *rs2* is *not* equal to zero, an idiom we refer to as SNEZ.

Finally, there are two special computational operations in RV32I, both of which use the U format. One is LUI, short for *load upper immediate*, which sets the upper 20 bits of register *rd* to the value of the 20-bit immediate, while setting the lower 12 bits of *rd* to zero. LUI is primarily used in conjunction with ADDI to load arbitrary 32-bit constants into registers, but it can also be paired with load and store instructions to access any static 32-bit address, or with an indirect jump instruction to transfer control to any static 32-bit address.

The other is AUIPC, short for *add upper immediate to* `pc`, which adds a 20-bit upper immediate to the `pc` and writes the result to register *rd*. AUIPC forms the basis for RISC-V's `pc`-relative addressing scheme: it is essential for reasonable code size and performance in position-independent code. To demonstrate this point, Figure 3.3 shows code sequences to load a variable that resides 0x1234 bytes away from the start of the code block, with and without AUIPC[5].

[3]A logical left shift by n is equivalent to multiplication by 2^n. A logical right shift by n is equivalent to unsigned division by 2^n, rounding towards zero, whereas an arithmetic right shift by n is equivalent to signed division by 2^n, rounding towards $-\infty$.

[4]There is no SUBI instruction, because ADDI with a negative immediate is almost equivalent. The one exception arises from the asymmetry of the two's complement representation: SUBI with an immediate of -2^{11} would add 2^{11} to a register, which ADDI is incapable of.

[5]Position-independent code without AUIPC could be implemented with a different ABI, rather than using the `jal` instruction to read the PC. The MIPS PIC ABI, for example, guarantees that `r25` always contains the address of the current function's entry point. But the effect is to move the extra instructions to the call site, since `r25` needs to be loaded with the callee's address.

```
auipc  x4, 0x1                jal  x4, 0x4
lw     x4, 0x234(x4)          lui  x5, 0x1
                              add  x4, x4, x5
                              lw   x4, 0x230(x4)

    WITH AUIPC                  WITHOUT AUIPC
```

Figure 3.3: Code fragments to load a variable 0x1234 bytes away from the `pc`, with and without the AUIPC instruction.

Memory Access Instructions

RV32I provides five instructions that load a value from memory into an integer register and three that store a value in a register to memory. All of these instructions use byte addresses to name memory locations; they form the address by adding the value in register *rs1* to the 12-bit sign-extended immediate. Table 3.3 lists all of the RISC-V memory operations.

We considered supporting additional addressing modes, including indexed addressing (i.e., *rs1*+*rs2*). However, this would have necessitated a third source operand for stores. Similarly, auto-increment addressing modes would have reduced instruction count, but would have added a second destination operand for loads. We could have employed a hybrid approach, providing indexed addressing only for some instructions and auto-increment for others, as did the Intel i860 [45], but we thought the extra instructions and non-orthogonality complicated the ISA. Additionally, we observed that most of the improvement in dynamic instruction count could be obtained by unrolling loops, which is typically beneficial for high-performance code in any case.

Misaligned loads and stores are explicitly allowed[6], but they are not guaranteed to execute atomically or with high performance. This caveat allows simple implementations to trap these instructions and emulate the misaligned access in low-level system software by non-atomically manipulating the adjacent words, but leaves the flexibility for higher-performance systems to implement them natively in hardware. (To date, all known RISC-V processors have employed the former strategy.) Other ISAs have approached this problem quite differently. The x86 requires that misaligned accesses execute atomically, effectively mandating they be implemented in hardware, at significant complexity cost. MIPS and Alpha both illegalized misaligned loads and stores, but provided additional instructions to handle the misaligned case. These consumed significant opcode space and, in the case of MIPS, added a new pipeline hazard. We reasoned that simply allowing misaligned accesses, but giving a great deal of flexibility to the implementation, was a better tradeoff.

[6]A load or store is considered to be misaligned if the address is not divisible by the width of the access. These operations can be complicated to implement in hardware because memories are usually organized in at least word-sized blocks, in which case misaligned accesses must be split in two. Worse, misaligned accesses can straddle cache line and page boundaries, the latter of which may affect the exception model.

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

Table 3.3: Listing of RV32I memory access instructions.

The load instructions all use the I-type instruction format. The LW instruction copies a 32-bit word from memory into integer register *rd*. LH and LB load 16-bit and 8-bit quantities, respectively, placing the result in the least-significant bits of *rd*, and filling the upper bits of *rd* with copies of the sign bit. LHU and LBU are similar, but instead they zero-fill the upper bits.

The stores are all S-type instructions. The SW instruction copies the 32-bit value in integer register *rs2* to memory. SH and SB copy the low 16-bits and 8-bits in *rs2* to half-word and byte-sized memory locations, respectively.

Memory Access Ordering

A RISC-V thread of execution perceives all of its own loads and stores to have occurred in program order, but in a multithreaded environment, there is no intrinsic guarantee of the order in which one thread perceives another thread's memory accesses. This design is referred to as a *relaxed memory model*. Weak memory models like RISC-V's are less intuitive than *sequential consistency* (SC), in which "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [59]. But SC effectively disallows several important memory system optimizations, like non-blocking loads and write buffering with bypassing [1].

Leveraging the observation that few memory ordering violations can actually become visible to other threads, out-of-order microarchitectures can speculate that reordering memory accesses is safe, and discard the incorrect execution if another thread might have been able to detect the reordering [106]. In effect, they can reuse their existing speculation mechanisms to give the appearance of SC but performance closer to that of a relaxed memory model [31]. Alas, this technique does not apply to simple, in-order microarchitectures, because they do not already have this expensive speculative execution hardware. Choosing SC as our memory

Instruction	Format	Meaning
`beq rs1, rs2, imm[12:1]`	SB	Branch if equal
`bne rs1, rs2, imm[12:1]`	SB	Branch if not equal
`blt rs1, rs2, imm[12:1]`	SB	Branch if less than, 2's complement
`bltu rs1, rs2, imm[12:1]`	SB	Branch if less than, unsigned
`bge rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, 2's complement
`bgeu rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, unsigned
`jal rd, imm[20:1]`	UJ	Jump and link
`jalr rd, rs1, imm[11:0]`	I	Jump and link register

Table 3.4: Listing of RV32I control transfer instructions.

model would have unduly penalized the performance of simpler RISC-V implementations.

Enforcement of memory ordering is hence made explicit in the ISA. RV32I provides a FENCE instruction that provides an ordering guarantee between memory accesses prior to the fence and subsequent to the fence. The arguments to the fence are two sets, the *predecessor* set and the *successor* set, which indicate what type of accesses are to be ordered by the fence: memory reads (R), memory writes (W), device input (I), and device output (O). For example, the instruction `fence rw,w` guarantees that all loads and stores prior to the fence will not appear to have executed before any store subsequent to the fence.

RV32I also provides an instruction to synchronize the instruction stream with data memory accesses, called FENCE.I. A store to instruction memory is only guaranteed to be reflected by subsequent instruction fetches after a FENCE.I has been executed. Some architectures, like x86, provide much stronger guarantees on the ordering between stores and instruction fetches. For systems with split instructions and data caches, the x86 design requires the two caches be kept coherent, e.g., by snooping the other cache on a miss. Since self-modifying code is relatively rare, we thought it best to allow simpler implementations with incoherent instruction caches, and instead place the onus on the programmers of self-modifying code to insert a FENCE.I instruction.

Control Flow Instructions

RV32I provides six instructions to conditionally change the flow of control, which Table 3.4 summarizes. These branch instructions, which use the SB-type instruction format, perform arithmetic comparisons between two registers and can transfer control to anywhere in a range of ±4 KiB (±1K instructions). The new address is formed by adding the sign-extended 12-bit immediate to the current `pc`. BEQ compares the values in registers *rs1* and *rs2* and takes the branch if they are equal. BLT compares *rs1* and *rs2* as two's complement integers and takes the branch if *rs1* is smaller. BLTU treats them as unsigned integers and takes the branch if *rs1* is smaller. BNE, BGE, and BGEU perform the same operations as BEQ, BLT, and BLTU, respectively, but have the opposite polarity.

A common feature of branches in other RISC architectures is the *branch delay slot*, in which the instruction immediately following the branch is executed whether or not the branch is taken. This design optimizes for shallow, single-issue, in-order pipelines, in which the branch direction would resolve in the same cycle that the subsequent instruction was being fetched. For that microarchitectural pattern, delay slots improve pipeline utilization and remove a control hazard. However, for deeper pipelines and for machines with superscalar instruction dispatch, delay slots increase complexity and offer vanishing benefit. In fact, they can even reduce performance: unfilled delay slots must be populated with a no-op, increasing code size[7]. Moreover, microarchitectural techniques can be used to keep pipelines busy without architecturally exposing a delay slot: branch target prediction can do so in the common case. (A two-entry branch target buffer, sufficient to capture most loop nests, adds about 128 bits of microarchitectural state, and so is easily justifiable for most pipelined implementations.)

In addition, to resolve branches early in the pipeline, many RISC architectures provide only simple branches. The Alpha, for example, only provides comparisons against zero; comparing two registers takes an additional instruction. Other ISAs, like SPARC, achieved this effect by only providing branches on condition code registers. In that case, many code sequences require an additional instruction to set the condition codes based upon a comparison. In RISC-V, we reasoned that we could achieve better code size and dynamic instruction count by folding the comparisons into the branch instruction. For pipelined implementations, this decision might require that branches be resolved in a later pipeline stage. But modern instruction pipelines tend to have accurate branch prediction and branch target prediction, so the slight increase in the taken branch latency should be more than balanced by the reduction in instruction count and code size.

We consciously omitted support for conditional moves and predication. Both enable some form of *if-conversion*, a transformation by which some control hazards can be traded for data hazards. Conditional move instructions are much weaker than predication: they add to the critical code path, and they cannot in general be used to if-convert instructions that might cause exceptions, like loads and stores. Full predication is much more general, but adds to the architectural state and consumes substantial opcode space, as each instruction must be given an additional predicate operand. Both techniques complicate implementations with register renaming, since the old value of the destination register must be copied to the new physical register when the predicate is false. Finally, if-conversion is usually not profitable in the common case that the condition is predictable: branch prediction will succeed, sometimes with higher performance, since it obviates the extra data dependence.

In addition to the branches, RISC-V provides two unconditional control transfer instruc-

[7]To improve performance in the face of unfilled branch delay slots, the MIPS and SPARC architects added new instructions that only execute the delay slot instruction if the branch is taken. Thus, the delay slot can always be filled for loop structures, since the first instruction of the loop can be copied into the delay slot. In this case, the code size penalty remains, since the instruction must in general be duplicated for the first loop iteration to execute correctly. Recognizing the deficiencies of branch delay slots, the MIPS architects finally made them optional in the 2014 revision of the ISA.

tions. The UJ-type JAL instruction, short for *jump-and-link*, sets the `pc` to anywhere in a ±1 MiB range (±256K instructions). It also writes the address of the instruction following the JAL (`pc`+4) to register *rd*. Thus, the instruction can be used for function calls, so that the callee knows the address to which to return. When linking to *rd* is not desired, as is the case for simple jumps, `x0` can be used for *rd*. We refer to this common idiom as the J pseudo-instruction. The RISC-V JAL instruction is `pc`-relative and so can be used freely in position-independent code[8].

Finally, the I-type JALR instruction provides an indirect jump to the value in register *rs1* plus a 12-bit sign-extended immediate. This versatile instruction is used for table jumps (as in C's `switch` statement), indirect function calls, and function returns. JALR discards the least-significant bit of the computed address, allowing software to store metadata in this bit and slightly reducing hardware cost. JALR was designed to be coupled with the AUIPC instruction to implement a two-instruction sequence for a `pc`-relative jump to anywhere in the 32-bit address space. As with JAL, the address of instruction following the JALR instruction (`pc`+4) is written to register *rd*.

All of the control-transfer instructions have two-byte granularity. This property is not intrinsically useful for RV32I code, wherein all instructions are four-byte aligned, but it enables instruction-set extensions whose instructions are any multiple of two bytes in length. One particularly important case, discussed in Chapter 5, is the compressed ISA extension, which adds 16-bit instructions to the ISA for improved code density.

System Instructions

Rounding out RV32I are the eight system instructions. Simple implementations may choose to trap these instructions and emulate their functionality in system software, but higher-performance implementations may implement more of their functionality in hardware. Table 3.5 lists these instructions.

The ECALL instruction is used to invoke the operating system to perform a system call. The RISC-V ISA itself does not define the parameter passing convention for system calls; it is a property of the ABI. The expectation is that most systems will pass parameters to system calls the same way that they pass parameters to normal function calls.

The EBREAK instruction is used to invoke the debugger. Unlike many ISAs, we did not allow for metadata to be encoded within the EBREAK instruction word. This feature would have consumed extra opcode space, but is not especially useful since the field cannot be wide enough to hold a complete memory address. Instead, this information is best stored in an auxiliary data structure, indexed by the program counter of the EBREAK instruction.

[8]In MIPS, by contrast, the J and JAL instructions are pseudo-absolute: the lower 28 bits of the new `pc` are provided directly from the immediate, whereas the upper bits come from the `pc` of the delay slot instruction. They are, as such, effectively useless in position-independent code; all unconditional control transfers either use the conditional branch instructions (if the target is nearby) or a load from the global offset table followed by an indirect jump.

SPARC wisely provided a `pc`-relative CALL instruction, but provided no corresponding JUMP instruction.

Instruction	Format	Meaning
`scall`	I	System call
`sbreak`	I	Breakpoint
`csrrw rd, csr, rs1`	I	Read and write CSR
`csrrc rd, csr, rs1`	I	Read and clear bits in CSR
`csrrs rd, csr, rs1`	I	Read and set bits in CSR
`csrrwi rd, csr, imm[4:0]`	I	Read and write CSR with immediate
`csrrci rd, csr, imm[4:0]`	I	Read and clear bits in CSR with immediate
`csrrsi rd, csr, imm[4:0]`	I	Read and set bits in CSR with immediate

Table 3.5: Listing of RV32I system instructions.

Name	Meaning
`cycle`	Cycle counter
`time`	Real-time clock
`instret`	Instructions retired counter
`cycleh`	Upper 32 bits of cycle counter
`timeh`	Upper 32 bits of real-time clock
`instreth`	Upper 32 bits of instructions retired counter

Table 3.6: Listing of RV32I control and status registers.

Six more instructions are provided to read and write *control and status registers* (CSRs), which provide a general facility for system control and I/O. The CSR address space supports up to 2^{12} control registers; presently, this space is only very sparsely populated. The CSRRW instruction copies the value in a CSR into integer register *rd*, and atomically overwrites the CSR with the value in integer register *rs1*. CSRRC atomically clears bits in a CSR. It copies the old value of a CSR to register *rd*, then for any bit set in register *rs1*, it atomically clears that bit in the CSR. CSRRS is similar, but sets bits in the CSR rather than clearing them. The remaining three instructions, CSRRWI, CSRRCI, and CSRRSI, behave like their counterparts without the letter I, but rather than taking the source operand from register *rs1*, they take it from a 5-bit zero-extended immediate.

Since reading or writing CSRs can have side effects, we define two special cases of these instructions, each of which explicitly lacks one of the side effects. CSRRS with *rs1*=`x0`, which reads a CSR and sets none of the bits in it, is defined to have no write side effects. This is the CSRR pseudo-instruction, which reads a CSR with no side effects. CSRRW with *rd*=`x0`, which writes a CSR but discards the old value, is defined to have no read side effects. This is the CSRW pseudo-instruction.

In most systems, the majority of CSRs are only accessible to privileged software, but RV32I does mandate a handful of user-level CSRs that provide a basic performance diagnostic facility. All are read-only so must be accessed using the CSRR psuedoinstruction. Table 3.6

```
retry:
    csrr  x3, cycleh
    csrr  x2, cycle
    csrr  x4, cycleh
    bne   x3, x4, retry
```

Figure 3.4: Sample code for reading the 64-bit cycle counter in RV32.

lists them. The `cycle` counter records the number of clock cycles that have elapsed since an arbitrary reference time. The `instret` counter ticks once per retired instruction. Finally, the `time` register reports a value proportional to elapsed wall-clock time. We provide both `cycle` and `time` because both quantities are figures of merit and are not necessarily linearly related. Many processor implementations use dynamic voltage and frequency scaling to modulate performance and power consumption as utilization and environmental constraints change over time, so the `time` register is necessary for basic performance measurement. On the other hand, the `cycle` counter is essential for diagnosing the performance of the processor pipeline.

Ideally, the `cycle`, `instret`, and `time` registers would have 64 bits of precision, because 32-bit counters overflow quickly: the `cycle` counter, for example, wraps after about one second in a 4 GHz implementation. In contrast, a 64-bit counter would take more than a century to overflow, presumably beyond the mean time to failure of the processor. To accommodate 64-bit counters in a 32-bit ISA, we provide additional CSRs, `cycleh`, `instreth`, and `timeh`, which contain the upper 32 bits of the corresponding counter. Of course, the act of reading both halves of one of the counters is not atomic, and the counter might overflow mid-sequence, particularly if an interrupt occurs. Figure 3.4 shows a scheme to correctly read the 64-bit cycle counter into the `x3`:`x2` register pair, obviating this concern.

3.2 The RV32E Base ISA

In low-end implementations of RV32I, the 31 integer registers are often the most expensive single hardware structure. Yet, for many embedded scenarios, a machine with substantially fewer registers would provide sufficient performance, and so the hardware cost is unjustifiable. Depending on the application, a design with fewer registers might also be more energy efficient. RV32E aims to cater to this domain.

RV32I and RV32E differ only in the number of integer registers: the latter reduces the count from 31 to 15. Figure 3.5 depicts the entirety of the user-visible architectural state in an RV32E machine. The RV32I instructions are all present in RV32E, and their behavior is the same, with the exception that any attempt to access the nonexistent registers `x16`–`x31` causes an exception. However, the performance counter CSRs mentioned in the previous section are optional in RV32E.

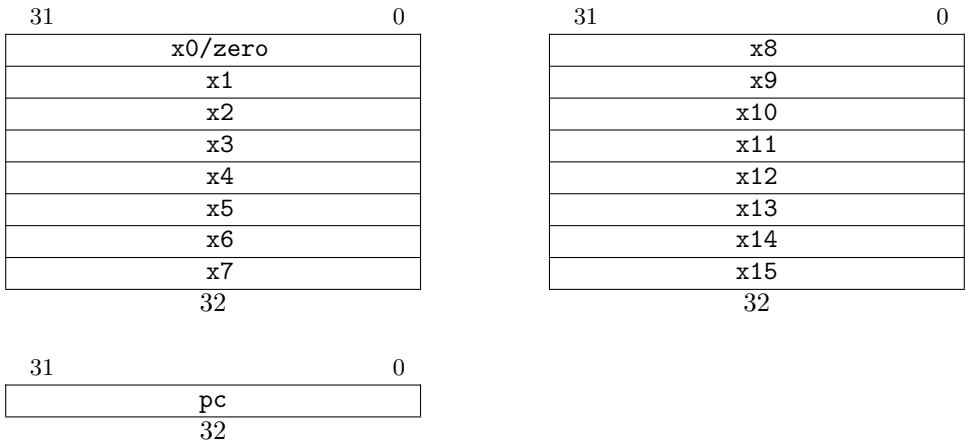

Figure 3.5: RV32I user-visible architectural state.

Since RV32E is only intended for deeply embedded scenarios, it is not meant to host a fully featured operating system environment. Hence, while RV32E is not ABI-compatible with RV32I, we are not concerned about the possibility of additional fragmentation in the RV32 software ecosystem.

3.3 The RV64I Base ISA

For an increasing number of application domains, 2^{32} bytes of addressable memory is insufficient. Large servers in 2015 have as much as 64 TiB of DRAM [84], requiring 46 bits to fully address. Even some wireless phones have exceeded 4 GiB of DRAM. Hence, while RV32I is appropriate for most small systems, its limited address space renders it unusable for many others. The RV64I base ISA addresses the lack of addresses.

As Figure 3.6 depicts, RV64I's user-visible state is very similar to RV32I's: it differs only in the widths of the integer registers and the program counter, which have all doubled in width to 64 bits. In the same spirit, the RV32I instructions perform the same function as in RV64I, except that they operate on the full 64-bit register. There are 12 new instructions in RV64I, as Table 3.7 lists.

While integer arithmetic often operates on the full width of a register, especially for the purpose of manipulating addresses, computations on sub-word quantities are also quite common. This effect is amplified in 64-bit architectures, since popular datatypes like `int` in Java and C remain 32 bits wide[9]. To maintain reasonable performance on 32-bit code, RV64I adds several computational instructions that operate on the lower 32 bits of the integer

[9]ISO C does not actually mandate that `int` be 32 bits wide, but so much software erroneously relies on this property that virtually all 64-bit C implementations have chosen not to widen the datatype to 64 bits.

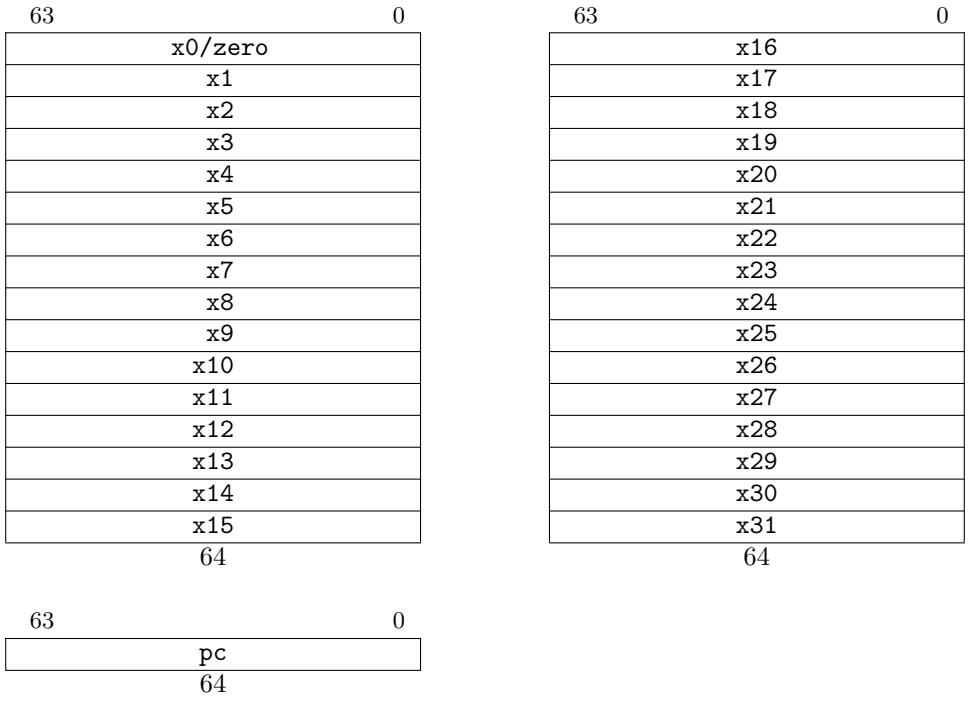

Figure 3.6: RV64I user-visible architectural state.

Instruction	Format	Meaning
`addw rd, rs1, rs2`	R	Add registers, 32-bit
`subw rd, rs1, rs2`	R	Subtract registers, 32-bit
`sllw rd, rs1, rs2`	R	Shift left logical by register, 32-bit
`srlw rd, rs1, rs2`	R	Shift right logical by register, 32-bit
`sraw rd, rs1, rs2`	R	Shift right arithmetic by register, 32-bit
`addiw rd, rs1, imm[11:0]`	I	Add immediate, 32-bit
`slliw rd, rs1, shamt[4:0]`	I	Shift left logical by immediate, 32-bit
`srliw rd, rs1, shamt[4:0]`	I	Shift right logical by immediate, 32-bit
`sraiw rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate, 32-bit
`lwu rd, imm[11:0](rs1)`	I	Load word, unsigned
`ld rd, imm[11:0](rs1)`	I	Load double-word
`sd rs2, imm[11:0](rs1)`	S	Store double-word

Table 3.7: Listing of additional RV64I computational instructions.

registers and produce 32-bit results, which are sign-extended to the full width of the register. These are the first nine instructions in Table 3.7.

Keeping 32-bit data sign-extended in the registers keeps casts between the C types `int` and `unsigned int`, and between `int` and `long`, costless operations. The existing branch instructions also automatically work on both signed and unsigned 32-bit types.

There are also three new memory access instructions: LWU loads a 32-bit word and zero-extends the result to 64 bits. LD and SD load and store 64-bit double-words.

3.4 The RV128I Base ISA

In 1976, Gordon Bell and Bill Strecker, two of the designers of DEC's hugely successful 16-bit PDP-11 architecture, keenly observed, "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" [20]. The *Virtual Address eXtension* to the PDP, better known by the acronym VAX, was less an extension of PDP than a complete redesign of the instruction set architecture: the PDP-11's opcode space was exhausted, and so the architecture could not be retrofitted to support a 32-bit address space.

Seeking to sidestep this pitfall, we consciously preserved a large fraction of RISC-V's opcode space for, among other possible extensions, a 128-bit address space variant, RV128I. While we expect 64 bits of address space to be ample for nearly all computing devices for decades to come, there are already plausible applications for a 128-bit address space, including single-address-space operating systems. The fastest supercomputer as of this writing, the Tianhe-2, has 1.3 PiB of memory, which would take 51 bits to completely byte-address. At the historic growth rate of the memory capacity of TOP500 champions, about 70% per year, a 64-bit address space would be exhausted in about two decades. To the extent that global addressability of such systems is desired, we contend that flat addressability is the best approach; RV128I provides a promising solution.

RV128I extends RV64I analogously to how RV64I extends RV32I: the width of the integer registers is doubled; new loads and stores are added; and the base arithmetic operations are redefined to operate on the full 128 bits. To maintain reasonable performance on 64-bit data, new arithmetic operations that process only the lower 64 bits are added. Table 3.8 summarizes the new instructions in RV128I.

3.5 Discussion

The RISC-V base ISAs are simple and straightforward to implement, yet complete enough to support a modern software stack. The base ISA design eschews architectural features that add undue complexity burdens to both simple and aggressive microarchitectures.

Nevertheless, there are many application domains for which a basic integer ISA cannot provide sufficient performance—for example, for workloads laden with floating-point compu-

Instruction	Format	Meaning
`addd rd, rs1, rs2`	R	Add registers, 64-bit
`subd rd, rs1, rs2`	R	Subtract registers, 64-bit
`slld rd, rs1, rs2`	R	Shift left logical by register, 64-bit
`srld rd, rs1, rs2`	R	Shift right logical by register, 64-bit
`srad rd, rs1, rs2`	R	Shift right arithmetic by register, 64-bit
`addid rd, rs1, imm[11:0]`	I	Add immediate, 64-bit
`sllid rd, rs1, shamt[5:0]`	I	Shift left logical by immediate, 64-bit
`srlid rd, rs1, shamt[5:0]`	I	Shift right logical by immediate, 64-bit
`sraid rd, rs1, shamt[5:0]`	I	Shift right arithmetic by immediate, 64-bit
`ldu rd, imm[11:0](rs1)`	I	Load double-word, unsigned
`lq rd, imm[11:0](rs1)`	I	Load quad-word
`sq rs2, imm[11:0](rs1)`	S	Store quad-word

Table 3.8: Listing of additional RV128I computational instructions.

tation. The next chapter describes four standard extensions to the base ISAs that improve the performance of RISC-V implementations in these domains.

Chapter 4

The RISC-V Standard Extensions

One of our goals in defining RISC-V was to make an ISA suitable for resource-constrained low-end implementations and high-performance ones alike. The former domain requires a lean base ISA, mandating that some features be left out—features that are important in *some* embedded processors and for any general-purpose applications processor. RISC-V provides such flexibility in the form of ISA extensions. This chapter describes four standard extensions—**M** for integer multiplication and division, **A** for atomic memory operations, and **F** and **D** for single- and double-precision floating-point—that, together, form a powerful ISA for general-purpose computing.

4.1 Integer Multiplication and Division

In many applications, particularly those rich in fixed-point computation, integer multiplication and division are common operations. Even when they do not represent a dominant fraction of computational operations, they can account for a substantial fraction of runtime when implemented with software subroutines. Hence, for most applications, hardware acceleration of these operations is desirable. Table 4.1 shows the instructions that the RISC-V **M** extension provides for this purpose.

We considered further separating the **M** extension into separate multiplication and division extensions, but reasoned that demand for these two extensions would be highly correlated. Furthermore, for low-end ASIC implementations with an iterative multiplier, an iterative divider can be added at relatively low incremental cost. On the other hand, for many FPGA implementations, division is quite a bit more expensive to implement than multiplication, thanks to the presence of hardened multiplier blocks; these processors may choose to trap division instructions and emulate them in software.

Unlike some RISC ISAs, which added special architectural registers for the operands to multiplication and division instructions, the instructions in RISC-V's **M** extension operate directly on the integer registers. This strategy reduces instruction count and latency by eliminating instructions that move to and from the special registers; enables superior compiler

Instruction		Format	Meaning
`mul`	`rd, rs1, rs2`	R	Multiply and return lower bits
`mulh`	`rd, rs1, rs2`	R	Multiply signed and return upper bits
`mulhu`	`rd, rs1, rs2`	R	Multiply unsigned and return upper bits
`mulhsu`	`rd, rs1, rs2`	R	Multiply signed-unsigned and return upper bits
`div`	`rd, rs1, rs2`	R	Signed division
`divu`	`rd, rs1, rs2`	R	Unsigned division
`rem`	`rd, rs1, rs2`	R	Signed remainder
`remu`	`rd, rs1, rs2`	R	Unsigned remainder
`mulw`	`rd, rs1, rs2`	R	Multiply, 32-bit
`divw`	`rd, rs1, rs2`	R	Signed division, 32-bit
`divuw`	`rd, rs1, rs2`	R	Unsigned division, 32-bit
`remw`	`rd, rs1, rs2`	R	Signed remainder, 32-bit
`remuw`	`rd, rs1, rs2`	R	Unsigned remainder, 32-bit

Table 4.1: Listing of RV32M and (below the line) RV64M instructions.

code scheduling; and reduces the size of the thread context. For some implementations, there is a slight increase in control logic for writeback arbitration, but we feel this minor cost is easily overshadowed by the benefits. For implementations with register renaming, this strategy actually reduces complexity, since it eliminates a class of registers that would otherwise need to be renamed to obtain reasonable performance.

RV32M adds four instructions that compute 32×32-bit products: `mul`, which returns the lower 32 bits of the product; and `mulh`, `mulhu`, and `mulhsu`, which return the upper 32 bits of the product, treating the multiplier and multiplicand as signed, unsigned, and mixed, respectively. (The lower 32 bits of the product do not depend on the signedness of the inputs.) The latter three instructions are essential for fixed-point computational libraries and enable an important strength reduction optimization: division by a constant can always be turned into multiplication by an approximate reciprocal, followed by a correction step to the upper half of the product [34].

There are also four instructions that perform 32-bit by 32-bit division: `div` and `divu`, for signed and unsigned division; and `rem` and `remu`, for signed and unsigned remainder. Following C99, signed division rounds toward zero, and the remainder has the same sign as the dividend. Division by zero does not cause an exception; programming languages that desire this behavior can branch on the divisor after initiating the division operation. This highly predictable branch should have little effect on performance.

Finally, RV64M extends these instructions to operate on the full 64 bits of the register, and also adds five more instructions, with the suffix `w`, that operate on 32-bit data and produce 32-bit sign-extended results.

4.2 Multiprocessor Synchronization

In 1962, Edsger Dijkstra famously observed that an algorithm conceived by Theodorus Dekker could solve the mutual exclusion problem using only regular loads and stores [28]. Alas, Dekker's algorithm scales poorly beyond two concurrent threads of execution: to acquire a lock shared between n threads takes $\mathcal{O}(n)$ operations. Consequently, the architects of early concurrent uniprocessors and parallel multiprocessors added hardware synchronization mechanisms to accelerate mutual exclusion, such as *test-and-set*, which atomically sets a bit of memory and returns the old value.

Mutual exclusion is a completely general synchronization mechanism, and the test-and-set approach is simple to implement. Nevertheless, this strategy also scales poorly to highly parallel systems, as it is insufficient to construct wait-free synchronization primitives, like non-blocking producer-consumer queues [40]. Several alternative primitives do suffice; of them, atomic *compare-and-swap* (CAS) is the most popular. CAS compares a register with a memory word, then atomically overwrites the memory word with a third datum if the comparands were equal.

Recognizing the resurgence of parallel computing, we sought to define RISC-V to scale to systems with a great degree of thread-level parallelism. It would have sufficed to make our memory atomics extension, named **A**, consist only of CAS; indeed, we considered this option. But CAS would have required a new integer instruction format with three source operands (memory address, comparand, and swap value), complicating processor microarchitectures and adding a new memory system command format with an additional data word. Instead, we followed the lead of several early RISC ISAs, including MIPS and Alpha, by providing *load-reserved* (LR) and *store-conditional* (SC) instructions. These lower-level primitives, conceived originally for Livermore Labs' S-1 project [22], split atomic operations into load, compute, and store phases. LR performs a normal load, but also registers a microarchitectural reservation on the memory address. The reservation may be forfeited if too much time elapses, or if another processor requests the same address. SC attempts to perform a store, but it succeeds only if the reservation is still held; additionally, it writes a success or failure code to an integer register so that software can branch on the result. Typically, an LR/SC sequence forms the body of a loop that iterates until the SC succeeds.

LR followed by a successful SC is an atomic operation on the modified memory word. In other words, no thread can observe another memory operation to have happened between the two. Similarly, an SC cannot be perceived to have occurred before its paired LR operation.

The principal virtue of the LR/SC scheme is that it is both straightforward to implement and quite general: it can be used to construct any single-word atomic operation, including CAS (see Figure 4.1). Unfortunately, naïve implementations of LR/SC suffer from livelock when multiple processors contend for the same data. To obviate this problem, we mandate that LR/SC sequences of bounded length (16 consecutive static instructions) will eventually succeed, provided they contain only base ISA instructions other than loads, stores, and taken

```
retry:  lr.w t0, (a0)           ## atomically {
        bne  t0, a1, fail       ##   if (M[a0] == a1)
        sc.w t0, a2, (a0)       ##     M[a0] = a2;
        bnez t0, retry          ## }
```

Figure 4.1: Compare-and-swap implemented using load-reserved and store-conditional.

branches[1]. This constraint allows an implementation to guarantee forward progress by simply holding off cache interventions for a bounded length of time. Of course, it also places some constraints on the microarchitecture. The instruction sequence must eventually fit in cache, even if it conflicts with the reserved datum; so, for example, unified instruction/data caches and TLBs must be at least two-way set-associative.

Another benefit of LR/SC over CAS is that it avoids the so-called ABA problem, in which a memory location is modified from the value A to the value B, then modified back to the value A. Because CAS success is determined by value equality, it cannot detect this scenario, complicating the implementation of wait-free data structures [27]. One common workaround is to provide a double-word CAS operation, where the second word serves as a version number. Alas, this approach complicates the ISA and the implementation: double-word CAS has four source and two destination register operands and modifies two memory words. Fortunately, LR/SC does not suffer from the ABA problem, since it detects any intervening memory write, whether or not the value was ultimately preserved.

Atomic Memory Operations

While it would have sufficed to provide only the LR/SC primitives, we also opted to include several atomic memory operations (AMOs), which perform simple arithmetic and logic operations on a memory word, then return the old value. The operations, which Table 4.2 summarizes, include addition; signed and unsigned minimum and maximum; bitwise AND, OR, and XOR; and swap. These AMOs enable an important optimization for highly parallel systems: when a memory word is contended, the AMO can be sent to the memory word, rather than obtaining exclusive access to the cache line containing the word [33]. In addition to reducing latency, network occupancy, and cache thrashing, this strategy ameliorates an Amdahl's law bottleneck. In contrast, it would be difficult to perform this optimization using LR/SC routines, since they comprise general instruction sequences. Of course, architects who do not wish to spare the hardware to implement AMOs directly can instead synthesize

[1]There are several subtleties to these constraints. The no-taken-branch constraint bounds the dynamic instruction count and, taken together with the 16-instruction limit, implies that direct-mapped instruction caches suffice—provided they can hold at least 64 consecutive bytes of any alignment. (The branch that retries the LR/SC sequence may, of course, be taken, but it must fit within the 16-instruction limit.) Disallowing other loads and stores in the sequence allows the use of direct-mapped data caches.

```
            andi  t0, a0, 3          # Shift
            slli  t0, t0, 3          #  addend
            sll   a1, a1, t0         #   into place.
            addi  t1, x0, 255        # Generate
            sll   t0, t1, t0         #  mask.
            andi  a0, a0, ~3         # Clear address LSBs.
retry:      lr.w  t2, (a0)           # Load operand.
            add   t3, a1, t2         # Perform
            xor   t4, t3, t2         #  addition
            and   t1, t4, t0         #   under
            xor   t2, t1, t2         #    mask.
            sc.w  t2, t2, (a0)       # Attempt to store result.
            bne   t2, x0, retry      # Retry if store fails.
```

Figure 4.2: Atomic addition of bytes, implemented with a word-sized LR/SC sequence.

them from the LR/SC primitives, either in microcode or with a software trap to a greater privilege level.

We consciously omitted AMOs on sub-word quantities, as they occur very rarely in most programs. The most common sub-word AMOs are bitwise logical operations, and these are readily implemented in terms of the word-sized AMOs, using a few additional instructions to mask the address and shift the datum. LR/SC sequences can realize the others. Figure 4.2 shows how an atomic fetch-and-add operation on a byte-sized operand might be implemented using LR/SC. The minimum and maximum operations may be similarly synthesized, taking care to use branchless sequences to implement their inherent conditional operations. These code sequences are grotesque, but they are invoked with such infrequency that they can be tucked away in subroutines.

The RVA Memory Model

As described in Section 3.1, RISC-V has a relaxed memory model, wherein one thread's memory accesses may be perceived in any order by another thread, unless a FENCE instruction is executed to guarantee a specific ordering. The instructions in the **A** extension have additional features that enable the efficient implementation of the *release consistency* (RC) memory model [30]. RC is a relaxed consistency model that allows a great degree of concurrency by distinguishing between different flavors of synchronization operations, assigning different, weaker ordering properties to each. RC has become the standard memory model for the 2011 revisions of the C and C++ languages, and it has long been compatible with Java's memory model.

In RC, shared memory accesses are divided into *ordinary* accesses—loads and stores whose reordering would not result in a race condition—and *special* accesses, which comprise

Instruction	Format	Meaning
`lr.{w|d} rd, (rs1)`	R	Load reserved
`sc.{w|d} rd, rs2, (rs1)`	R	Store conditional
`amoswap.{w|d} rd, rs2, (rs1)`	R	Atomic swap
`amoadd.{w|d} rd, rs2, (rs1)`	R	Atomic addition
`amoand.{w|d} rd, rs2, (rs1)`	R	Atomic bitwise AND
`amoor.{w|d} rd, rs2, (rs1)`	R	Atomic bitwise OR
`amoxor.{w|d} rd, rs2, (rs1)`	R	Atomic bitwise XOR
`amomin.{w|d} rd, rs2, (rs1)`	R	Atomic two's complement minimum
`amomax.{w|d} rd, rs2, (rs1)`	R	Atomic two's complement maximum
`amominu.{w|d} rd, rs2, (rs1)`	R	Atomic unsigned minimum
`amomaxu.{w|d} rd, rs2, (rs1)`	R	Atomic unsigned maximum

Table 4.2: Listing of RVA instructions. The instructions with the `w` suffix operate on 32-bit words; those with the `d` suffix are RV64A-only instructions that operate on 64-bit words.

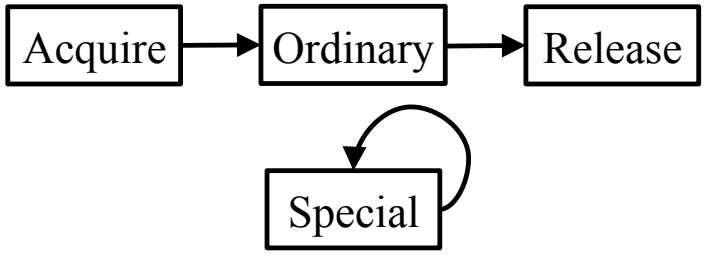

Figure 4.3: Orderings between accesses mandated by release consistency. The origin of an arrow cannot be perceived to have occurred before the destination of the arrow.

all others. RC introduces two important types of special access: *acquire* and *release*. Acquire operations are used to gain access to shared variables, whereas release operations grant access to others. They are frequently associated with mutex acquisition and relinquishment, respectively. The semantics of the acquire and release primitives is given by the memory ordering constraints in the RC model, which Figure 4.3 depicts. Before ordinary loads and stores are allowed to perform, all previous acquires must have been performed. Likewise, before a release is allowed to perform, all previous ordinary loads and stores must have performed. Finally, special accesses must be totally ordered.

The latter constraint results in a stricter form of RC, *RCsc*, in which special accesses are sequentially consistent. This crucial property enables the *data-race-free* programming model [2], a contract between the hardware and software that guarantees the appearance of a sequentially consistent memory system, provided that all special accesses are appropriately annotated. The effect is that, for an important class of programs, RCsc provides both the greater concurrency of relaxed models and the simpler programming model of sequential consistency.

The instructions in the **A** extension realize the RCsc memory model with a memory ordering annotation, which comprises two bits on the static instruction: *aq* and *rl*. When the *aq* bit is set on an LR, SC, or AMO, the instruction acts as an acquire access, so that it cannot be perceived to have executed *after* any subsequent memory access issued by the same thread. When the *rl* bit is set, the instruction acts as a release access, so that it cannot be perceived to have executed *before* any prior memory access in that thread. When neither *aq* nor *rl* is set, no particular ordering is guaranteed, which is appropriate for associative reductions. When both are set, the access is defined to be sequentially consistent with respect to other such accesses[2].

4.3 Single-Precision Floating-Point

Floating-point computation is ubiquitous in some application domains, and even many integer-oriented programs use enough floating-point to justify hardware support. Nevertheless, we excluded floating-point instructions from the base ISA to support embedded implementations that would make no use of them, and for deployments where floating-point arithmetic is best handled by attached coprocessors. RISC-V's **F** extension adds single-precision floating-point support, compliant with the 2008 revision of the IEEE 754 standard for floating-point arithmetic [7].

The most contentious decision in defining the **F** extension was whether to reuse the integer registers for floating-point computation, or to add dedicated floating-point registers. The former strategy would have simplified the ISA, resulted in lower-cost implementations, and reduced context switch time. Nevertheless, we ultimately chose to add a new set of floating-point registers, shown in Figure 4.4, as we felt the pros outweighed the cons:

- The natural widths for the integer and floating-point registers are not necessarily the same—for example, an RV32 machine might have double-precision floating-point.
- We wanted to grant implementations the flexibility to employ an internal *recoded* format, e.g., to accelerate the handling of subnormal numbers in hardware. Not representing integers and floating-point numbers in the same registers simplifies such a design.
- A split register file organization increases the total number of registers addressable from a single instruction, because the opcode (floating-point versus integer) provides an implicit register specifier bit.

[2]We expressly define that setting both *aq* and *rl* bits results in a sequentially consistent access, because it is not inherently so. Together, acquire and release impose a total order on the accesses from that thread—but not on all accesses globally. In other words, two threads could perceive different interleavings of each other's stream of accesses, which would violate sequential consistency. Fortunately, the addition of the *store atomicity* property, imposed by single-writer cache coherence protocols [16], implies that *aq*+*rl*=*sc*. Therefore, in practice, our additional restriction comes at no performance cost.

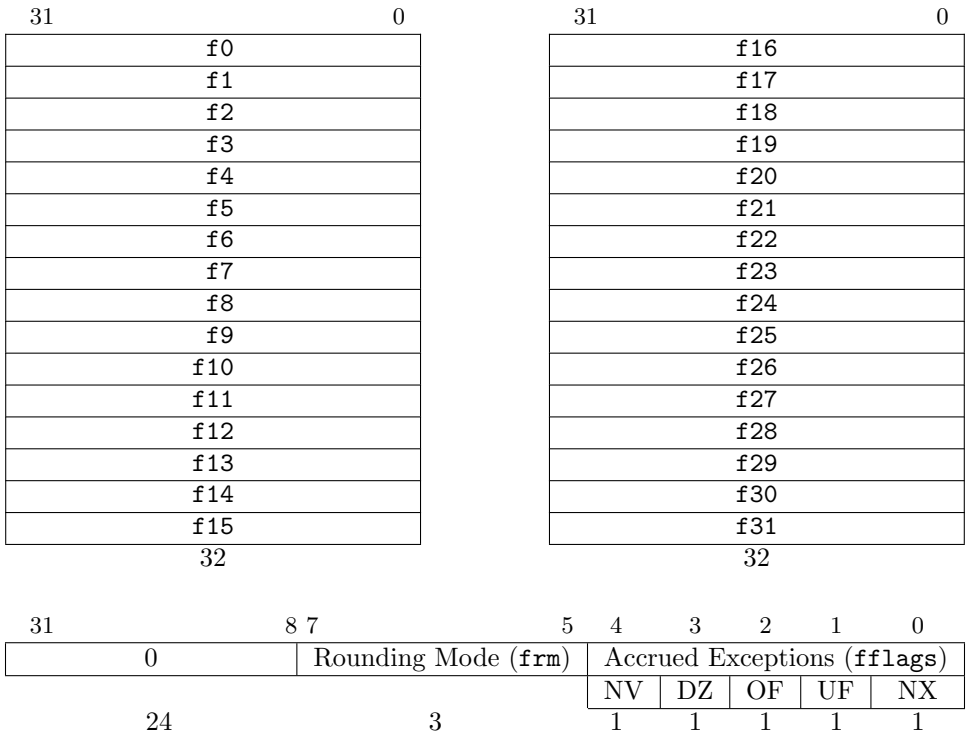

Figure 4.4: RVF user-visible architectural state.

- A split organization provides a natural register file banking strategy, simplifying the provision of register file ports for superscalar implementations.
- The context switch cost can be mitigated by adding microarchitecturally managed dirty bits to the register file.

Concordant with the desire to support an internal recoded format, we also chose to avoid representing integers in the floating-point register file at all. Unlike SPARC, Alpha, and MIPS, which perform conversions to and from fixed-point entirely within the floating-point register file, RISC-V uses the integer register file for the integral operands to these instructions. This choice also shortens common instruction sequences for mixed-format code—for example, when using the result of a conversion to integer in an address calculation.

Most floating-point operations round their results, and the desired rounding scheme varies by programming language and algorithm. We provide five rounding modes, the encoding of which Table 4.3 shows: rounding to the nearest number, with ties broken towards the even number; rounding towards zero; rounding towards $-\infty$; rounding towards $+\infty$; and rounding to the nearest number, with ties broken by rounding away from zero. (IEEE 754-2008 requires the only the first four, but in our experience, the fifth can be useful for the hand-coding of library routines.)

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
111	DYN	*Dynamic rounding, based on* `frm` *register*

Table 4.3: Supported rounding modes and their encoding.

In most programming languages, the rounding direction is expected to be specified dynamically. Accordingly, all floating-point instructions can use a dynamic rounding mode, set in the `frm` field of the floating-point control and status register, the format of which Figure 4.4 shows. Additionally, some operations, like casts from floating-point to integer types in C and Java, require rounding in a specific direction. Supporting such operations without modifying the dynamic rounding mode is desirable, and also serves to speed up the implementation of library routines, like the transcendental functions. So, we give all floating-point operations the option of using the dynamic rounding mode or choosing one of the rounding modes statically. This additional three-bit field causes the **F** extension to consume a substantial amount of opcode space, but we felt the generality and improved performance justified the expense.

Exception Handling

Most floating-point operations can generate what the IEEE 754 standard refers to as *exceptions*: runtime conditions that indicate arithmetic errors or imprecision. The five exceptions are invalid operation (raised by such operations as $\sqrt{-1}$); division by zero; overflow; underflow; and inexact (indicating that rounding occurred). The standard does not mandate that these exceptions cause traps, and in RISC-V, we chose not to translate these exceptions into traps to facilitate non-speculative out-of-order completion of floating-point operations. Had we chosen otherwise, implementors of in-order pipelines would be forced to choose between imprecise traps, which expose the implementation and complicate system software, and in-order completion, which either deepens the integer pipeline or reduces performance.

Nevertheless, it remains possible to write software that takes action on floating-point exceptions. The five exceptions accrue into the floating-point status register; software can examine this register and branch to a user-level exception handler based upon its value. Since raising the accrued exception flags is an associative operation, providing the accrued exceptions register still permits out-of-order instruction completion. Instructions that directly manipulate the exception flags need simply interlock until all outstanding instructions complete.

ISA	Sign	Significand	QNaN Polarity
SPARC	0	11111111111111111111	1
MIPS	0	01111111111111111111	0
PA-RISC	0	01000000000000000000	0
x86	1	10000000000000000000	1
Alpha	1	10000000000000000000	1
ARM	0	10000000000000000000	1
RISC-V	0	10000000000000000000	1

Table 4.4: Default single-precision NaN for several ISAs. QNaN polarity refers to whether the most significant bit of the significand indicates that the NaN is quiet when set, or quiet when clear. The values come from [87, 67, 54, 47, 3, 8].

NaN Generation and Propagation

The IEEE 754-1985 floating-point standard [6] forced a compromise between several industrial competitors who, collectively, sought to put an end to what Velvel Kahan described as "anarchy" in floating-point arithmetic [52]. (For some participants, the less altruistic motivation was to keep Intel from running away with the standard.) To minimize dissatisfaction amongst the factions, the standard left several details up to the implementation—for example, whether underflow is detected before or after rounding, how NaNs are generated and propagated, and how signaling NaNs are distinguished from quiet NaNs[3].

The leeway granted by the standard has led to a surprising degree of fragmentation. Table 4.4 lists the default NaN encoding for seven ISAs, which between them have five distinct encodings! Two ISAs, MIPS and PA-RISC, chose to represent quiet NaNs with the quiet bit clear, which violates the 2008 revision of the IEEE 754 standard[4]. The others differ in whether the rest of the significand is set or clear, and, perhaps more surprisingly, whether the sign bit is set. For RISC-V, we chose a scheme where the sign bit is clear and all significand bits except the quiet bit are clear, for four reasons:

- It is the same default NaN as at least one other ISA (ARM), so our choice does not exacerbate the fragmentation of IEEE 754 implementations.

[3]Signaling NaNs are those that cause the Invalid Operation exception to be raised when consumed. Because signaling NaNs are never *generated* by floating-point instructions—they must be supplied as input to a computation—they remain a very rarely used, but required, feature of the IEEE 754 standard.

[4]The original IEEE 754-1985 standard permitted the scheme used by the MIPS and PA-RISC, but it quickly became apparent that it was a poor design choice. If the quiet bit has positive polarity, then a signaling NaN can be converted to a quiet NaN simply by setting the quiet bit. But for these designs with negative polarity, merely clearing the quiet bit might turn a signaling NaN into infinity, rather than a quiet NaN. So, more logic is required for this conversion. For designs that propagate NaN payloads, the range of the payload is effectively reduced.

- It is the same as the Java programming language's canonical NaN[5]. (Most other programming languages do not define one.)
- There are 2^{22} quiet NaN patterns, but only $2^{22} - 1$ signaling NaN patterns. Our choice of canonical NaN is the only quiet NaN that cannot be generated by quieting a signaling NaN. Thus, for systems that choose to propagate NaN payloads, *generated* NaNs can be distinguished from *propagated* signaling NaNs.
- Clearing most significand bits has lower hardware cost than setting most significand bits. Some functional units already require a datapath to supply zeros in the significand, but do not have the corresponding datapath to supply all ones.

The standard also provides the option to propagate the NaN *payload*, i.e., the significand bits below the quiet bit, from a NaN input to the output of an operation. This feature was intended to preserve diagnostic information, such as the origin of the NaN. For three reasons, we chose *not* to provide this feature in RISC-V:

- The NaN payload is too small to hold a complete memory address, so it is difficult to use the feature to encode meaningful diagnostic information.
- Because NaN payload propagation is optional in the standard, it cannot be relied upon by portable software, so the feature is rarely used.
- Propagating the NaN payload increases hardware cost.

Instead, whenever a NaN is emitted by a computational instruction, we require it be the canonical NaN. Implementors are, of course, free to provide a NaN payload propagation scheme as a non-standard extension, enabled by a non-standard processor mode bit, but our default scheme remains mandatory.

Floating-Point Instructions

Table 4.5 lists the 30 new instructions that the **F** extension introduces. They are divided into four categories: data movement instructions, conversions, comparisons, and arithmetic instructions. Among the new data movement instructions are new loads and stores, FLW and FSW, which move data between memory and the floating-point register file. Also included are instructions that move floating-point values between the floating-point and integer register files, FMV.X.S and FMV.S.X. Some ISAs, like SPARC and (initially) Alpha, omitted these instructions, requiring instead that a memory temporary be used. Their design removes a pipeline hazard but can substantially increase instruction count for mixed-format code. Figure 4.5 demonstrates this effect for a function that computes the integer logarithm of a

Instruction	Format	Meaning
`flw fd, imm[11:0](rs1)`	I	Load 32-bit floating-point word
`fsw fs2, imm[11:0](rs1)`	I	Store 32-bit floating-point word
`fmv.x.s rd, fs1`	R	Move from floating-point to integer register
`fmv.s.x fd, rs1`	R	Move from integer to floating-point register
`fsgnj.s fd, fs1, fs2`	R	Inject sign
`fsgnjn.s fd, fs1, fs2`	R	Inject complement of sign
`fsgnjx.s fd, fs1, fs2`	R	Multiply signs
`fcvt.w[u].s rd, fs1`	R	Convert to [un]signed 32-bit integer
`fcvt.s.w[u] fd, rs1`	R	Convert from [un]signed 32-bit integer
`fcvt.l[u].s rd, fs1`	R	Convert to [un]signed 64-bit integer (RV64)
`fcvt.s.l[u] fd, rs1`	R	Convert from [un]signed 64-bit integer (RV64)
`feq.s rd, fs1, fs2`	R	Set if equal
`flt.s rd, fs1, fs2`	R	Set if less than
`fle.s rd, fs1, fs2`	R	Set if less than or equal
`fmin.s frd, fs1, fs2`	R	Compute minimum of two values
`fmax.s frd, fs1, fs2`	R	Compute maximum of two values
`fclass.s rd, fs1`	R	Classify floating-point value
`fadd.s fd, fs1, fs2`	R	Add two registers
`fsub.s fd, fs1, fs2`	R	Subtract two registers
`fmul.s fd, fs1, fs2`	R	Multiply two registers
`fdiv.s fd, fs1, fs2`	R	Divide two registers
`fsqrt.s fd, fs1`	R	Compute square root
`fmadd.s fd, fs1, fs2, fs3`	R4	`fs1`×`fs2`+`fs3`
`fmsub.s fd, fs1, fs2, fs3`	R4	`fs1`×`fs2`−`fs3`
`fnmadd.s fd, fs1, fs2, fs3`	R4	−(`fs1`×`fs2`+`fs3`)
`fnmsub.s fd, fs1, fs2, fs3`	R4	−(`fs1`×`fs2`−`fs3`)

Table 4.5: Listing of RVF instructions.

floating-point number by extracting its exponent. The variant without FMV.X.S needs 60% more instructions and may suffer a data cache miss.

A novel feature of the **F** instructions are the *sign-injection* instructions, which copy the exponent and significand from one number, but generate a new sign. FSGNJ copies the sign from a second number; FSGNJN copies the negated sign from a second number; and FSGNJX takes the new sign from the product of the two input signs. When the two source operands are the same, these instructions can be used to move, negate, or take the absolute value of a register. In their more general form, they are useful for the hand-coding of floating-point library routines.

The conversion instructions change between integer and floating-point formats. Integer

[5]We note with some amusement that Sun, the creators of the Java programming language, chose a different canonical NaN for Java than for their ISA, SPARC.

```
fmv.x.s a0, fa0
srli    a0, a0, 23
andi    a0, a0, 0xff
addi    a0, a0, -127
jalr    x0, ra, 0
```

WITH FMV.X.S

```
addi  sp, sp, -16
fsw   fa0, 0(sp)
lw    a0, 0(sp)
srli  a0, a0, 23
andi  a0, a0, 0xff
addi  a0, a0, -127
addi  sp, sp, 16
jalr  x0, ra, 0
```

WITHOUT FMV.X.S

Figure 4.5: A routine that computes $\lfloor \log_2 |x| \rfloor$ by extracting the exponent from a floating-point number, with and without the FMV.X.S instruction.

sources and destinations use the integer register file, which improves performance for mixed-format code by eliminating explicit moves between the two register files. Not representing integers in the floating-point register file also has the benefit of simplifying the implementation of an internal recoded floating-point format. The four conversion instructions convert to and from signed integers (FCVT.W.S and FCVT.S.W) and unsigned integers (FCVT.WU.S and FCVT.S.WU), using any of the supported rounding modes.

The comparison instructions set an integer register with the Boolean result of equality and inequality tests. FEQ.S compares two floating-point numbers for equality; FLT.S performs a less-than comparison; and FLE.S performs a less-than-or-equal comparison. The results are always false on NaN inputs. FLT and FLE raise an invalid exception on quiet NaN inputs, so behave like the C operators `<` and `<=`; FEQ doesn't, so behaves like the C operator `==`. To branch on the result of a floating-point comparison, an integer BEQ or BNE on the Boolean result is used.

Two additional comparison instructions are FMIN.S and FMAX.S, which compute the minimum and maximum of two floating-point numbers, respectively. They implement the IEEE 754 operations *minNum* and *maxNum*, which do not propagate NaNs: if one input is NaN, the other is returned. FMIN's behavior conveniently matches the C library routine `fminf`. Alas, it does not match the common idiom `(a < b ? a : b)`, which always evaluates to `a` if either `a` or `b` is NaN.

The final floating-point comparison instruction is the floating-point classify instruction, FCLASS.S, which writes to an integer register a mask categorizing the floating-point number. Table 4.6 lists these classes. Since the mask is one-hot encoded, it is possible to test membership of multiple classes with a single ANDI instruction. FCLASS is useful for the hand-coding of library routines, since they often branch on the unusual inputs, like NaNs. The alternative approach of moving the number to the integer register file and deconstructing it into its constituent fields takes many more instructions.

The arithmetic instructions include the usual suspects: addition, subtraction, multipli-

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 4.6: Classes into which the FCLASS instruction categorizes Format of result of FCLASS instruction.

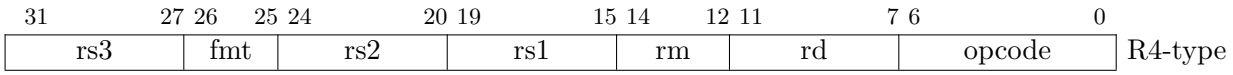

Figure 4.6: Fused multiply-add instruction format, R4.

cation, division, and square root. As required to support IEEE 754-2008 with reasonable efficiency, we also provide four fused multiply-add (FMA) instructions, which compute any of the four operations $\pm a \times b \pm c$, without an intermediate rounding of the product. This property allows these instructions to accelerate the implementation of some floating-point library routines, and in general can improve the performance and accuracy of many algorithms. For programs with approximate parity between the frequency of floating-point addition and multiplication, FMA substantially reduces the effort necessary to achieve high throughput as compared to an architecture with only discrete multiply and add. Consider, for example, the dense linear algebraic operation `C += A×B`, for 4×4 matrices stored in memory. Without FMA, this operation takes 192 instructions, including 272 floating-point register reads and 176 writes. With FMA, it takes 128 instructions, including 208 reads and 112 writes—a 33% reduction in instruction traffic and 29% reduction in operand traffic.

4.4 Double-Precision Floating-Point

Most general-purpose systems need both single- and double-precision floating-point, partially because single-precision is not sufficiently precise for some algorithms and partially because modern programming languages have a bias towards double-precision arithmetic. Nevertheless, we felt that it was best to separate the **F** extension from the **D** extension, be-

cause in some embedded domains, single-precision floating-point suffices and double-precision floating-point is too expensive. The popular ARM Cortex-M4 [15], a 32-bit microcontroller, embodies this design point, as do myriad digital signal processors.

The **D** extension is structured very similarly to the **F** extension, and indeed requires the presence of the **F** extension. The 32 floating-point registers are doubled in width to 64 bits, and new instructions are added that operate on double-precision values. 32-bit and 64-bit floats cannot be freely mixed in computational instructions, but instructions to convert between the formats (FCVT.D.S and FCVT.S.D) are provided to support mixed-format code.

We contemplated an alternative design wherein the registers remain 32 bits wide and double-precision values are stored in aligned register pairs, as was the case in SPARC and MIPS. However, this design is less suitable for implementations with register renaming, wherein the two halves of the value may no longer be physically adjacent. More importantly, this design would have halved the number of architectural registers available to double-precision routines, reducing the effectiveness of register blocking and increasing instruction count. Expanding the register widths seemed to be the best alternative, despite the extra 1024 bits of architectural state.

One caveat of this approach is that there is no facility to move double-precision floating-point numbers to and from the integer register file in RV32. We contemplated adding three instructions for this purpose. Two of them, FMVHI.X.D and FMVLO.X.D, would have copied the upper and lower halves of a floating-point number to an integer register. The third, FMV.D.X, would have formed a double-precision float from two integer registers and moved that value to a floating-point register. Since the latter would have been the only floating-point instruction with two integer sources, we thought it best to drop support for these operations altogether and require the use of a memory temporary. The more common case of conversions to integer datatypes is still supported for RV32D.

Table 4.7 summarizes the instructions in the **D** extension.

4.5 Discussion

Together with one of the base RVI ISAs, the **M**, **A**, **F**, and **D** extensions provide a sufficient basis for general-purpose scalar computation, and so we collectively term them **G**. While many specific applications would benefit from the addition of a few new instructions not included in **G**, we think it is unlikely that we excluded any particular scalar instruction that would be be broadly beneficial.

RVG code is performant, but its fixed 32-bit encoding is not particularly compact. The next chapter proposes another RISC-V ISA extension, **C**, which seeks to improve the density of RVG code by providing a compressed 16-bit encoding for the most common instructions.

Instruction	Format	Meaning
`fld fd, imm[11:0](rs1)`	I	Load 64-bit floating-point word
`fsd fs2, imm[11:0](rs1)`	I	Store 64-bit floating-point word
`fmv.x.d rd, fs1`	R	Move from F.P. to integer register (RV64)
`fmv.d.x fd, rs1`	R	Move from integer to F.P. register (RV64)
`fsgnj.d fd, fs1, fs2`	R	Inject sign
`fsgnjn.d fd, fs1, fs2`	R	Inject complement of sign
`fsgnjx.d fd, fs1, fs2`	R	Multiply signs
`fcvt.w[u].d rd, fs1`	R	Convert to [un]signed 32-bit integer
`fcvt.d.w[u] fd, rs1`	R	Convert from [un]signed 32-bit integer
`fcvt.l[u].d rd, fs1`	R	Convert to [un]signed 64-bit integer (RV64)
`fcvt.d.l[u] fd, rs1`	R	Convert from [un]signed 64-bit integer (RV64)
`fcvt.s.d fd, fs1`	R	Convert from double to single
`fcvt.d.s fd, fs1`	R	Convert from single to double
`feq.d rd, fs1, fs2`	R	Set if equal
`flt.d rd, fs1, fs2`	R	Set if less than
`fle.d rd, fs1, fs2`	R	Set if less than or equal
`fmin.d frd, fs1, fs2`	R	Compute minimum of two values
`fmax.d frd, fs1, fs2`	R	Compute maximum of two values
`fclass.d rd, fs1`	R	Classify floating-point value
`fadd.d fd, fs1, fs2`	R	Add two registers
`fsub.d fd, fs1, fs2`	R	Subtract two registers
`fmul.d fd, fs1, fs2`	R	Multiply two registers
`fdiv.d fd, fs1, fs2`	R	Divide two registers
`fsqrt.d fd, fs1`	R	Compute square root
`fmadd.d fd, fs1, fs2, fs3`	R4	`fs1`×`fs2`+`fs3`
`fmsub.d fd, fs1, fs2, fs3`	R4	`fs1`×`fs2`−`fs3`
`fnmadd.d fd, fs1, fs2, fs3`	R4	−(`fs1`×`fs2`+`fs3`)
`fnmsub.d fd, fs1, fs2, fs3`	R4	−(`fs1`×`fs2`−`fs3`)

Table 4.7: Listing of RVD instructions.

Chapter 5

The RISC-V Compressed ISA Extension

This chapter describes a standard extension to the RISC-V ISA called RISC-V Compressed, or RVC for short. RVC introduces dual-length instructions to the base ISA, aiming to reduce the static code size and dynamic fetch traffic of RISC-V programs by encoding the most frequent instructions in a denser format. The smaller instruction footprint reduces the cost of embedded systems, for which instruction storage is a significant cost, and improves performance for cache-based systems by reducing instruction cache misses. Fetching fewer bytes from instruction memory can significantly reduce energy dissipation, of which instruction delivery can be a dominant fraction [69].

RVC was originally proposed in my Master's thesis [99] as an extension to the original RISC-V 1.0 ISA. Since then, RISC-V 2.0 has been finalized and a new ABI has been adopted, rendering RVC 1.0 obsolete. In this chapter I describe an updated version of RVC that has been adapted to RISC-V 2.0 and re-engineered to achieve greater code density, regularize the ISA, and reduce the complexity of instruction decoding.

5.1 Background

The first stored-program computers used fixed-length instruction encodings [105, 58] but were quickly followed by machines that, in an effort to reduce the cost of the program store, employed variable-length instruction sets. The IBM Stretch [21], introduced in 1961, had both 32-bit and 64-bit instruction formats, wherein some of the 64-bit instructions could also be encoded as 32-bit instructions, depending on the size of operands and the registers referenced. The CDC 6600 [95] and Cray-1 [82] ISAs, in many respects the precursors to RISC, each had two lengths of instruction, albeit without the redundancy property of the Stretch.

The designers of the first RISC architectures [78, 56, 67] favored regular, 32-bit fixed-width instruction encodings that were straightforward to decode and performed only simple

operations. In contrast, the CISC minicomputers of the era [89] employed microcoded control and so could readily support complex instructions of variable length, using fewer bits to encode common operations with fewer operands and simpler addressing modes. The RISC approach enabled low-cost, high-performance single-chip implementations, but their design came at the cost of code density. A program might have taken more RISC instructions to encode than CISC ones, and the RISC ones were typically at least as wide; thus, it took quite a bit more storage to represent a typical program on a RISC than on a CISC. The RISC-I architects, for example, observed a 50% code size increase relative to the DEC VAX and PDP-11 architectures [78].

In high-end machines, the performance impact of loose RISC instruction encodings was mitigated as the bounty of Moore's Law reduced the cost of integrated instruction caches. In the domain of embedded systems, however, limited instruction memory capacity and bandwidth often disfavored RISC architectures. To expand their markets, RISC vendors like MIPS and ARM created variants of their ISAs, named MIPS16 and Thumb, that could be encoded in narrower 16-bit fixed-width instructions [68, 11]. Most of these instructions were equivalent to an existing instruction or sequence thereof in the base ISA, with restrictions on register access patterns and operand sizes[1].

Although these compressed RISC ISAs did substantially improve code size, they had several disadvantages. Foremost, the base ISAs were not designed with their compressed counterparts in mind. Opcode space had not been reserved for the compressed variants, and so the only way to accommodate them was to make new, incompatible instruction sets. This precluded the free intermixing of base ISA instructions. In MIPS16, for example, ISAs could only be swapped with special jump instructions, and so MIPS and MIPS16 code was only mixed at procedure-call boundaries.

The unavailability of the base ISA instructions had significant performance implications. Some operations that could easily be encoded in a single 32-bit instruction, like loading a large constant, might have taken as many as three 16-bit instructions. The extra intermediate results exacerbated register pressure in ISAs that already had severely limited register sets. Moreover, encoding an entire ISA in 16 bits meant that important functionality, like floating-point, had to be left out. Later compressed RISC ISA variants, such as microMIPS [65] and Thumb-2 [12], corrected this deficiency and allowed 16-bit and 32-bit instructions to be freely intermixed. Alas, the 32-bit instructions were still not the same as those in the base ISA, forcing implementers to design and verify two instruction decoders, increasing hardware cost, and vastly complicating the software ecosystem.

[1]For example, of the 31 general-purpose registers in the MIPS ISA, most MIPS16 instructions can only access eight. Other instructions implicitly reference one of the other 23 registers. The `move` instruction, which can access any register, serves as an escape hatch.

5.2 Implications for the Base ISA

With the benefit of hindsight, we designed RISC-V from the ground-up to seamlessly support dual-length instructions. Crucially, the base ISA and standard extensions occupy only a small fraction of their opcode space. Unlike earlier RISC ISAs that densely populated their 32-bit encoding space—SPARC, for example, spent $\frac{1}{4}$ of its opcode space on its CALL instruction alone [88]—the base RISC-V encoding consumes less than $\frac{1}{4}$ of its 32-bit opcode space. We consciously reserved the remaining $\frac{3}{4}$ of this space to encode a compressed ISA extension[2].

Additionally, although the base ISA consists of only 32-bit instructions that must be naturally aligned, the base control-flow instructions have a displacement granularity of 16 bits. This design allows for ISA extensions that contain instructions whose length is any multiple of two bytes, without adding new branches and jumps to the base ISA. We note that it would have been straightforward to further support instructions with arbitrary byte alignment. We reasoned, however, that 16-bit instructions would like obtain most of the potential savings, and so the incremental benefit of 8- and 24-bit instructions would not likely justify the increased hardware complexity in the instruction fetch units. More importantly, this decision would have decreased the range of branches and jumps even further, thus increasing instruction count and offsetting some of the code size and performance improvements of the new instruction widths.

The only change to the base ISA to support RVC, then, is to relax the alignment restriction on the base ISA instructions and allow them to begin on any 16-bit boundary. Obviously, there would have been some benefit to keeping the alignment constraint in place, as it would have simplified the instruction fetch hardware. But doing so would have forfeited too much of the code density improvement. Suppose, for example, that $\frac{1}{2}$ of instructions were compressible and that they were distributed uniformly. Without the alignment requirement, the RVC code would then be $\frac{1}{4}$ smaller than the RISC-V code. But $\frac{1}{3}$ of the remaining 32-bit instructions would become misaligned as a result[3], and, had we not relaxed the alignment requirement, would require padding. Padding is most readily achieved by not compressing the immediately previous instruction, effectively reducing the fraction of RVC instructions from $\frac{1}{2}$ to $\frac{1}{3}$ and reducing the code size savings from $\frac{1}{4}$ to $\frac{1}{6}$. Although this effect could be mitigated to some extent by code scheduling, the additional constraints on the compiler would expose more data hazards and reduce performance, particularly for statically scheduled implementations.

[2]Another important consequence of this encoding choice is that it is very easy to detect whether an instruction is 16 or 32 bits in length: only two opcode bits need be examined. This scheme greatly accelerates superscalar instruction decoding, which requires that instructions be serially scanned to determine their boundaries.

[3]Misalignment results from a 32-bit-wide instruction following an odd number of 16-bit-wide instructions. Assuming uniform distribution of 16-bit instructions, which represent a fraction p of total instructions, the probability that a 32-bit instruction will be misaligned is $p(1-p)+p^3(1-p)+p^5(1-p)+\ldots=\frac{p}{1+p}$, or $\frac{1}{3}$ for $p=\frac{1}{2}$. So, while the fractional code size reduction is $\frac{p}{2}$ without the alignment constraint, it is $\frac{p^2}{1+p}$ with the constraint.

5.3 RVC Design Philosophy

Two major design constraints guided RVC's design. First, RVC programs should take no more instructions to encode than their RISC-V counterparts and so should be at least as performant[4]. This goal is easily achieved with a modeless design in which the base ISA instructions are always available. An important consequence of this design is that RVC need not be a standalone ISA, and so precious RVC opcode space need not be spent on essential but relatively infrequent operations, like invoking the operating system with a system call. Instead, that opcode space can be dedicated to reducing the size of the most common code sequences.

Second, each RVC instruction must expand into a single RISC-V instruction. The reasons for this constraint are twofold. Most importantly, it simplifies the implementation and verification of RVC processors: RVC instructions can simply be expanded into their base ISA counterparts during instruction decode, and so the backend of the processor can be largely agnostic to their existence. Furthermore, assembly language programmers and compilers need not be made aware of RVC: code compression can be left to the assembler and linker[5]. This constraint does, however, preclude some important code size optimizations: notably, *load-multiple* and *store-multiple* instructions, a common feature of other compressed RISC ISAs, do not fit this template. Section 5.6 discusses the implications of their omission.

Given these constraints, the ISA design problem reduces to a simple tradeoff between compression ratio and ease of instruction decode cost. At one extreme, we could map each of the available RVC opcodes to the most common RISC-V ones, perhaps even using a per-program dictionary [61]. While this approach would achieve the greatest compression, it has three principal drawbacks. The dictionary lookup is costly, offsetting the instruction fetch energy savings. It also adds significant latency to instruction decode, likely reducing performance and further offsetting the energy savings. Finally, the dictionary adds to the architectural state, increasing context switch time and memory usage.

Fortunately, four properties of typical RISC-V instruction streams render such a general approach unnecessary:

- *Instructions express great spatial locality of register reference.* RISC-V provisions an ample register set to minimize register spills and facilitate register blocking, but even so, most accesses are to a small number of registers. Figure 5.1 shows the static frequency of register reference in the SPEC CPU2006 benchmarks, sorted by register class. Three special registers—the zero register `x0`, the link register `ra`, and the stack pointer `sp`—collectively account for one-quarter of all static register references. The first argument register `a0` alone accounts for one-sixth of references.

[4]Indeed, RVC programs often take fewer instructions to encode than RISC-V ones. The smaller code reduces `pc`-relative offsets, reducing the number of branches that displace more than 2 KiB and jumps that displace more than 1 MiB, which require an extra instruction to encode.

[5]The RISC-V port of the GNU C Compiler has been programmed to favor the register subset that most RVC instructions may address, but is otherwise ignorant of RVC's existence. Of course, making the compiler more RVC-aware could avail more opportunities for code compression.

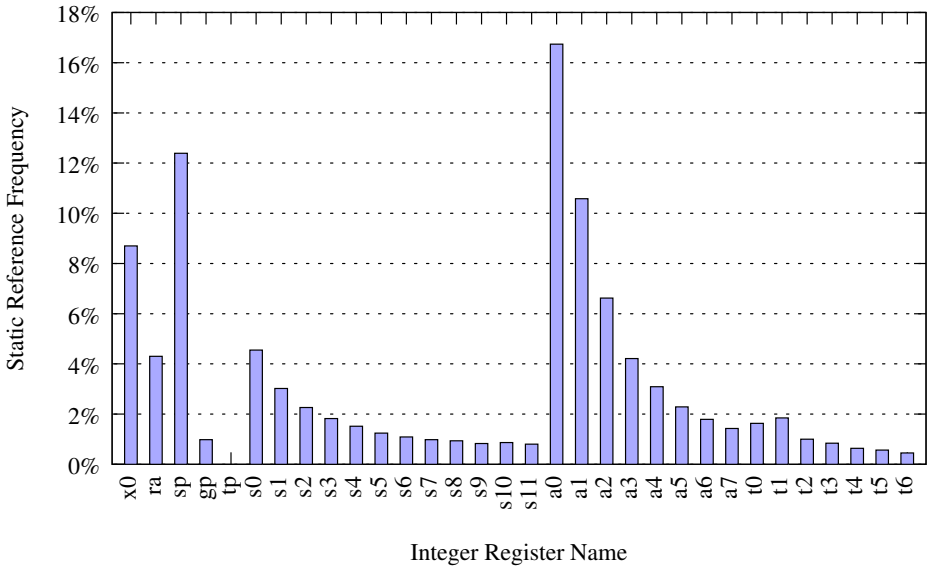

Figure 5.1: Frequency of integer register usage in static code in the SPEC CPU2006 benchmark suite. Registers are sorted by function in the standard RISC-V calling convention. Several registers have special purposes in the ABI: `x0` is hard-wired to the constant zero; `ra` is the link register to which functions return; `sp` is the stack pointer; `gp` points to global data; and `tp` points to thread-local data. The `a`-registers are caller-saved registers used to pass parameters and return results. The `t`-registers are caller-saved temporaries. The `s`-registers are callee-saved and preserve their contents across function calls.

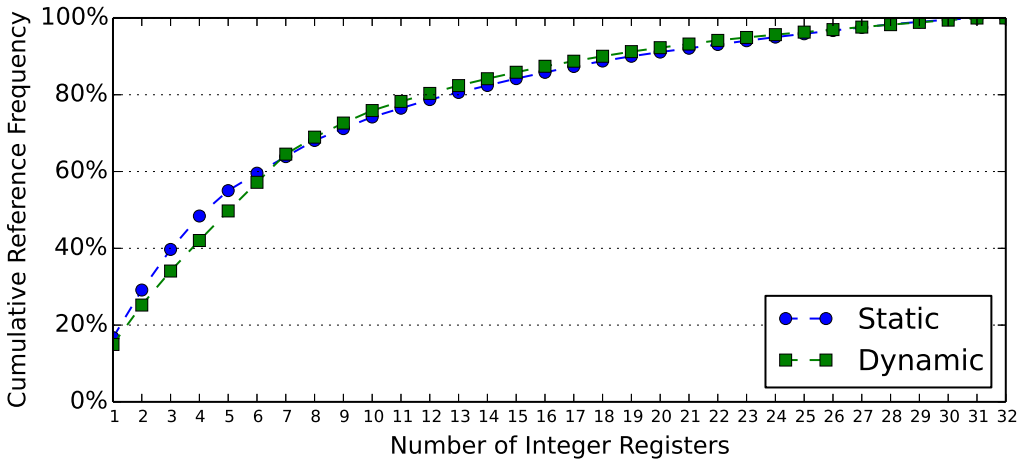

Figure 5.2: Cumulative frequency of integer register usage in the SPEC CPU2006 benchmark suite, sorted in descending order of frequency.

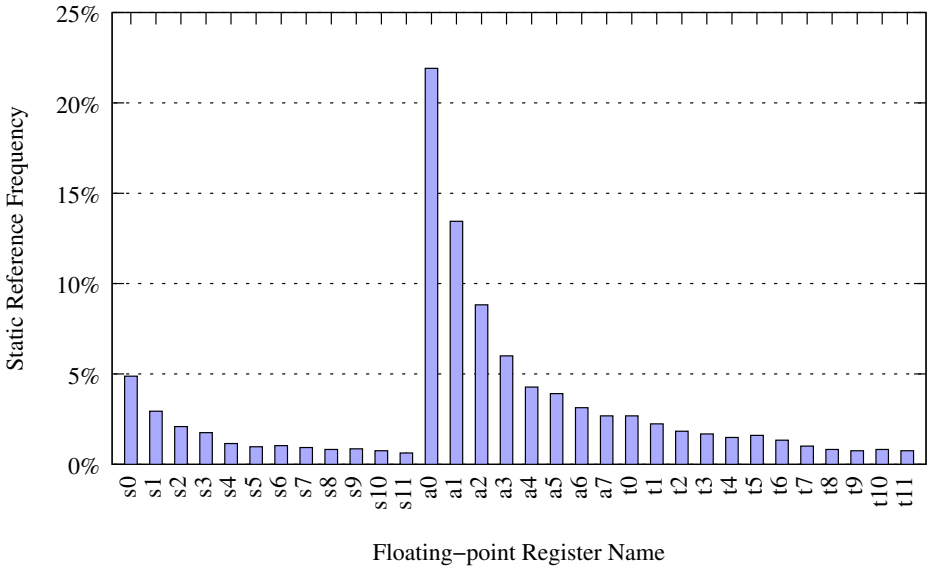

Figure 5.3: Frequency of floating-point register usage in static code in the SPEC CPU2006 benchmark suite. Registers are sorted by function in the standard RISC-V calling convention. Like the integer registers, the `a`-registers are used to pass parameters and return results; the `t`-registers are caller-saved temporaries; and the `s`-registers are callee-saved.

Figure 5.2 renders the static and dynamic frequencies of register reference as a cumulative distribution function. Notably, just one-quarter of the registers account for two-thirds of both static and dynamic references.

Statically, floating-point register accesses exhibit a similar degree of locality (see Figures 5.3 and 5.4), with the exception that there are no special-purpose registers to exploit. Dynamically, an even smaller number of registers dominates: the eight most commonly used registers account for three-quarters of accesses.

- *Instructions tend to have few unique operands.* Some instruction sets, such as the Intel 80x86, provide only destructive forms of many operations: one of the source operands is necessarily overwritten by the result. This ISA decision substantially increases instruction count for some programs, and so RISC-V provides non-destructive forms of all register-register instructions. Nevertheless, destructive arithmetic operations are common: 47% of static arithmetic instructions in SPEC CPU2006 share at least one source operand with the destination register.

  Additionally, the degenerate forms of several RISC-V instructions express common idioms. For example, the ubiquitous ADDI instruction is used to synthesize small constants when its source register is `x0`, or to copy a register when its immediate is 0.

- *Immediate operands and offsets tend to be small.* Roughly half of immediate operands can be represented in five bits (see Figure 5.5).

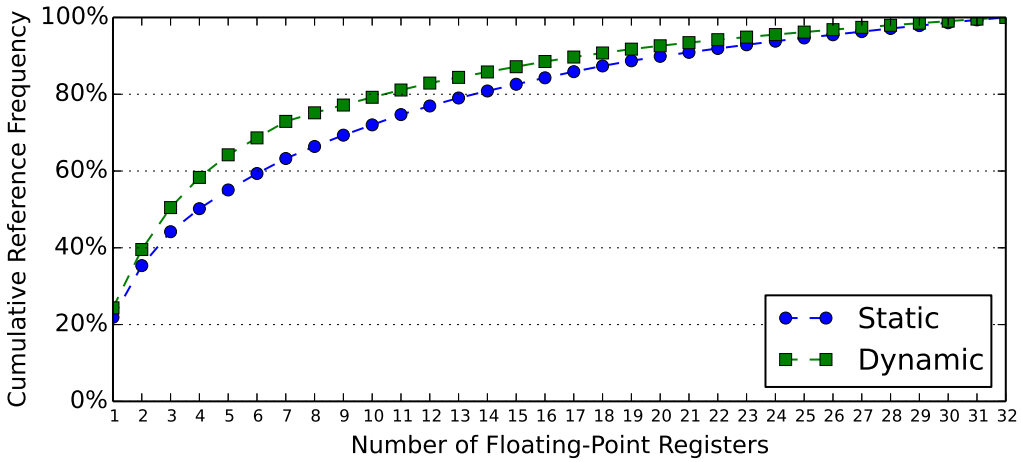

Figure 5.4: Cumulative frequency of floating-point register usage in the SPEC CPU2006 benchmark suite, sorted in descending order of frequency.

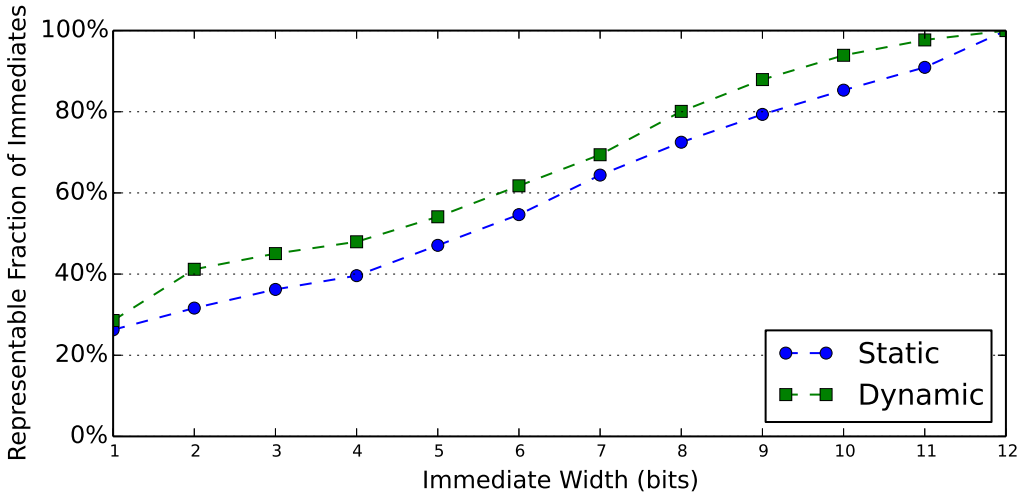

Figure 5.5: Cumulative distribution of immediate operand widths in the SPEC CPU2006 benchmark suite when compiled for RISC-V. Since RISC-V has 12-bit immediates, the immediates in SPEC wider than 12 bits are loaded with multiple instructions and manifest in this data as multiple smaller immediates.

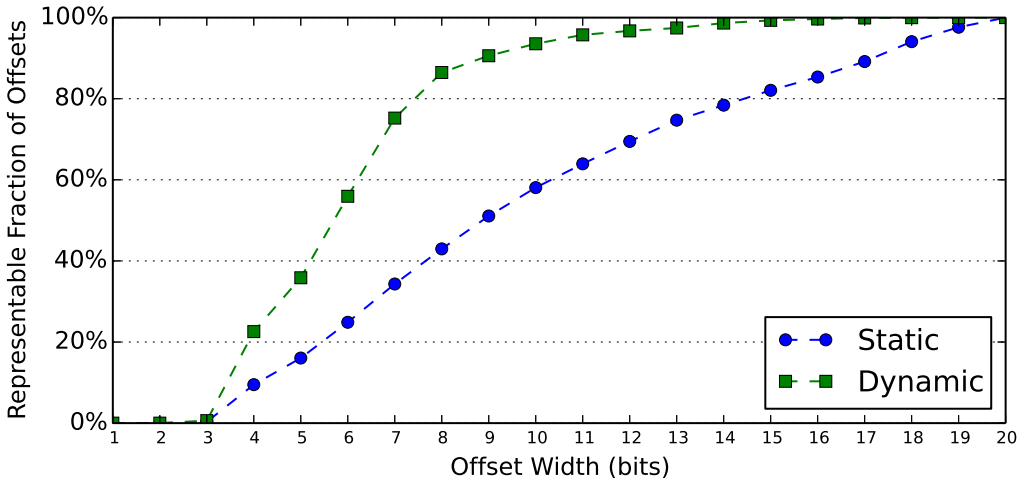

Figure 5.6: Cumulative distribution of branch offset widths in the SPEC CPU2006 benchmark suite. Branches in the base ISA have 12-bit two's-complement offsets in increments of two bytes ($\pm 2^{10}$ instructions). Jumps have 20-bit offsets ($\pm 2^{18}$ instructions).

Figure 5.6 shows the distribution of branch and jump offsets. Statically, branch and jump offsets are often quite large; dynamically, however, almost 90% fit within eight bits, reflecting the dominance of relatively small loops. Furthermore, since this data was collected from uncompressed RISC-V programs, the branches and jumps displace up to twice as much as they would in RVC programs. Effectively, for RVC programs, the CDFs would translate to the left by up to one bit.

- *A small number of unique opcodes dominates.* The vast majority of static instructions in SPEC CPU2006 (74%) are integer loads, adds, stores, or branches. As Table 5.1 indicates, the twenty most common RISC-V opcodes account for 91% of static instructions and 76% of dynamic instructions. The most common instruction, ADDI, accounts for almost one-quarter of instructions statically and one-seventh dynamically.

Leveraging these observations, we propose a design for RVC that can express the most common forms of the most frequent instructions, while preserving encoding regularity so as to simplify the implementation.

5.4 The RVC Extension

The RISC-V Compressed ISA extension is encoded in the three-quarters of the 16-bit RISC-V encoding space not occupied by the base ISA. We partition this space into 24 major opcodes, each of which can encode 11 bits of operands. Some of the major opcodes are

Instruction	Static Frequency	Cumulative	Instruction	Dynamic Frequency	Cumulative
ADDI	23.23%	23.23%	ADDI	14.36%	14.36%
LD	12.87%	36.10%	LD	8.27%	22.63%
SD	8.13%	44.23%	FLD	6.83%	29.46%
JAL	7.73%	51.96%	ADD	6.23%	35.69%
ADD	5.71%	57.67%	LW	4.38%	40.07%
LW	5.06%	62.73%	SD	4.29%	44.36%
LUI	3.69%	66.42%	BNE	4.14%	48.50%
FLD	3.00%	69.42%	SLLI	3.65%	52.15%
BEQ	2.96%	72.39%	FMADD.D	3.49%	55.64%
SLLI	2.96%	75.34%	BEQ	3.27%	58.91%
SW	2.93%	78.27%	ADDIW	2.86%	61.77%
ADDIW	2.33%	80.60%	FSD	2.24%	64.00%
BNE	2.32%	82.92%	FMUL.D	2.02%	66.02%
JALR	1.68%	84.60%	LUI	1.56%	67.59%
FSD	1.56%	86.16%	SW	1.52%	69.10%
BGE	1.15%	87.31%	JAL	1.38%	70.49%
FMUL.D	0.84%	88.15%	BLT	1.37%	71.86%
FMADD.D	0.84%	88.99%	ADDW	1.34%	73.19%
BLT	0.83%	89.82%	FSUB.D	1.28%	74.47%
ADDW	0.77%	90.59%	BGE	1.27%	75.75%

Table 5.1: Twenty most common RV64IMAFD instructions, statically and dynamically, in SPEC CPU2006. ADDI's outsized popularity is due not only to its frequent use in updating induction variables but also to its two idiomatic uses: synthesizing constants and copying registers.

further subdivided to encode instructions that require fewer operand bits or are not common enough to justify a larger operand encoding space.

RVC Operand Encoding

Table 5.2 lists the major RVC instruction formats; several minor formats differ in the encoding of immediate operands. The CR, CI, and CSS formats can address all 32 registers and are generally reserved for the most common operations, like copying registers or accessing the stack.

Given that a few integer registers are accessed far more often than their peers, RVC conserves encoding space by granting many instructions access only to the most popular ones. The CIW, and CL instructions can access only eight, `x8`–`x15`, called the *RVC registers*. We selected an aligned block of eight registers to minimize decoding circuitry[6]. In the standard

[6]The choice of `x8`–`x15` might seem arbitrary, but had we selected `x16`–`x23` or `x24`–`x31` instead, the RVC

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
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

Table 5.2: Major RVC instruction formats, from [101].

calling convention, these correspond to two callee-saved registers, `s0` and `s1`, and six caller-saved argument registers, `a0`–`a5`.

Additionally, instructions in several of the formats implicitly access registers with special significance in the ABI: the zero register `x0`, the link register `x1` (`ra`), and the stack pointer `x2` (`sp`). This decision increases decoding complexity but is necessary to capture several common idioms, like register spills, within a reasonably sized encoding space. The cost of the decoding circuitry is mitigated by the fact that all implicit registers' identifiers have their three most significant bits in common (they are all zero).

Most instruction formats contain an immediate operand. Unfortunately, the most common immediate values vary considerably between instructions. For example, while some RISC-V instructions commonly have negative immediates, others rarely do. Induction variable decrements and backwards branches are typical examples of the former case. Stack-relative loads and stores, on the other hand, never have negative offsets because the ABI illegalizes references to the part of the stack that lies below the stack pointer[7]. Hence, unlike in the base ISA, which has only sign-extended immediates, some RVC instructions have zero-extended ones. Had we opted for sign-extended immediates only, 12% fewer loads and stores would have been compressible.

Similarly, nearly all loads and stores in RISC-V programs are naturally aligned, in which case their offsets are divisible by the word size. Accounting for this property, all load and store offsets in RVC are scaled by the word size. Had we not done so, 44% fewer loads and stores would have been compressible. Had we additionally sign-extended these immediates, the loss would have grown to 62%—or 19% fewer opportunities for compression overall.

extension would be effectively incompatible with RV32E (the embedded ISA variant with a limited integer register set).

[7]Stacks grow downwards in the standard calling convention. This convention is somewhat arbitrary, but it is now codified in the RVC ISA.

RVC Instructions

Table 5.3 lists the instructions in RV32C and RV64C, and the base ISA instruction to which they expand. The ISAs have a combined total of 44 instructions, of which 39 are valid in RV64C and 32 are valid in RV32C.

Integer arithmetic operations are the most common class of instruction, accounting for 19 opcodes. Five RVC instructions expand to ADDI alone, reflecting its frequent idiomatic usage: increment (C.ADDI), increment stack pointer (C.ADDI16SP), generate stack-relative address (C.ADDI4SPN), load immediate (C.LI), and C.NOP[8]. Many of the arithmetic instructions are destructive and can target only RVC registers; the most frequently used ones can target any register.

Loads and stores are the next-most represented class of instruction, accounting for 16 opcodes. RV32C can move 32-bit integers and floats and 64-bit floats to and from memory; RV64C supports 32-bit and 64-bit integers but only 64-bit floats. (The RV32C 32-bit floating-point loads and stores occupy the same opcode space as the RV64C 64-bit integer loads and stores.) For each data type, two addressing modes are provided: base-plus-displacement, where the base address comes from one of the eight RVC registers, and stack-pointer-relative, with a wider displacement. The former must use an RVC register for the load or store data, whereas the latter may use any. In all cases, the displacements are unsigned and scaled by the data size.

Control-flow instructions round out the ISA. RVC provides conditional branches that test for equality with zero (C.BEQZ and C.BNEZ), an unconditional direct jump (C.J), and a register-indirect jump (C.JR). Additionally, forms of the latter two instructions that link to `x1`, C.JAL and C.JALR, are suitable for function calls in the standard calling convention. (Limited opcode space precluded the inclusion of C.JAL in RV64C.) C.EBREAK is a breakpoint instruction, simplifying the debugging of RVC programs.

RVC Instruction Encoding

Table 5.4 provides the complete encoding of the RV64C instruction set. Reflecting their ubiquity, loads and stores occupy 50% of the encoding space. Of the remaining space, arithmetic instructions take up three-quarters, and the control-flow instructions consume the rest.

An immediately evident feature of this encoding scheme is the multitude of immediate operand encoding formats, a consequence of the cramped encoding space. As in the base ISA, however, the immediate encoding is scrambled to minimize the cost of generating the immediate operand. Despite the twelve immediate choices, eight of the 18 immediate bits always come from the same bit positions in the instruction; five come from only two positions; four come from three positions; and one comes from four positions.

[8]Unlike the base ISA, where the idiom for copying a register, MV, maps to ADDI, the RVC instruction C.MV expands to ADD. This is because there is no RVC instruction format that can encode both a 5-bit *rd* and a 5-bit *rs1*, but there is one that can encode *rd* and *rs2*.

There are several other subtleties to this encoding. For the instructions with sign-extended immediates, the sign bit always resides in the same position, bit 12. Furthermore, these instructions all lie in a different opcode-space quadrant from those with zero-extended immediates; hence, the most-significant immediate bits, 18 and above, can be generated from just three instruction bits.

Decoding the register specifiers is also simpler than a cursory glance might suggest. With the exception of instructions that access the implicit registers `x0`, `x1`, and `x2`, the register specifiers each come from one of at most three positions, counting the position within the base RISC-V ISA encoding; decoding which one requires examining just three opcode bits. Nevertheless, decoding the register specifiers is on the critical path for many implementations, particularly superscalars, which must analyze the data hazards between the instructions in an issue packet. Aggressive implementations might require additional pipelining to cope with the increased latency, or additional cross-check logic and register map table ports to speculatively decode all combinations of register specifiers.

As Table 5.5 shows, many opcodes are reserved for potential future use. The instruction `0x0000`, which would otherwise map to a redundant instruction[9], is permanently reserved to improve the likelihood of trapping errant code. Additionally, despite the temptation to use all of the opcode space to maximize compression on current code, one major opcode and several subminor opcodes have been reserved in case RVC fails to capture important patterns in future software. (4.6% of the RV64C encoding space has yet to be allocated). Finally, besides the canonical no-op, all instructions that don't modify architectural state (e.g., increment a register by zero) are reserved for future microarchitectural hints. On implementations for which the hints are meaningless, these instructions will correctly perform no operation, at no additional hardware cost.

5.5 Evaluation

We measured the efficacy of RVC at compressing RISC-V code by comparing the static code size and dynamic instruction fetch traffic of RVC and RISC-V versions of several programs: the SPEC CPU2006 benchmark suite [91], nominally representative of workstation workloads; Dhrystone [104], a historical synthetic benchmark of questionable relevance but outsized popularity; CoreMark [29], an embedded systems benchmark; and the Linux kernel, version 3.14.29 [94]. We compiled the programs with the RISC-V port of GCC 5.2.0 and used an assembler and linker modified to select the RVC variants of instructions whenever possible. The userland programs were dynamically linked against the C and FORTRAN runtime libraries[10]. Static compression ratios were computed by dividing the size of the RVC binaries'

[9] `C.ADDI4SPN rd', 0`, if legal, would perform the same operation as `C.MV rd, sp`.

[10] We chose not to statically link against the C library because it is far larger than most of the benchmarks, and so it has a strongly homogenizing effect on the static instruction mix. We wished to see how effectively RVC compresses a variety of programs, and so including the C library would have defeated this purpose. Of course, the dynamically linked C library still contributes to the dynamic instruction mix, and this is reflected

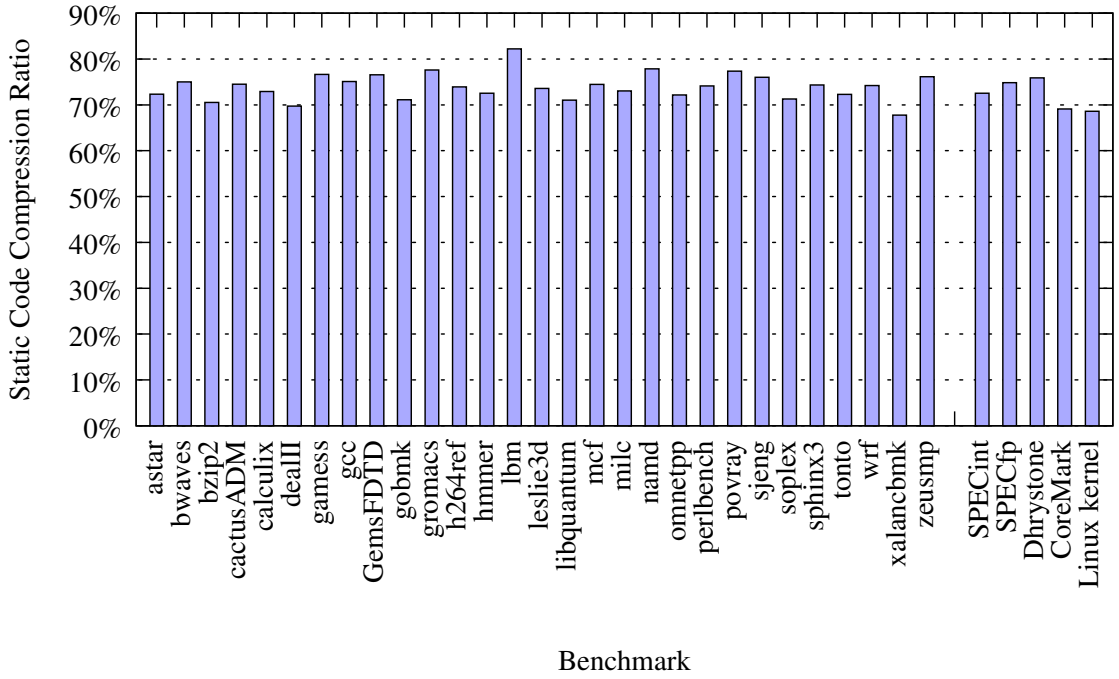

Figure 5.7: Static compression of RVC code compared to RISC-V code in the SPEC CPU2006 benchmark suite, Dhrystone, CoreMark, and the Linux kernel. The SPECfp outlier, `lbm`, is briefly discussed in the next section.

text segments by that of their RISC-V counterparts. To compute the dynamic compression ratios, we ran the programs on `spike` [100], the RISC-V ISA simulator, to obtain the dynamic instruction mix. We used the reference input set for SPEC and the default settings for Dhrystone and CoreMark. For Linux, we measured the kernel boot procedure, including the execution of the `init` process, the `ash` shell, and the `poweroff` command.

Static Code Compression

Figure 5.7 shows the static compression ratios for the programs in the SPEC CPU2006 benchmark suite. Compression ratios range from 68.8% to 82.2%. For SPECint, we see a geometric mean compression ratio of 73.7%, i.e., 52.6% of RISC-V instructions were compressed. For SPECfp, we see a slightly worse compression ratio of 74.1%, owing to the underrepresentation of floating-point instructions in RVC. Dhrystone and CoreMark shrink to 75.9% and 69.1% of their original sizes, respectively.

The Linux kernel, easily the most important benchmark in this set, compresses substantially: the RVC kernel is 68.6% of the size of the RISC-V one. Interestingly, a small fraction

in our dynamic measurements.

The RVC C library's text is 27.6% smaller than RISC-V's.

of this savings is attributable to a 0.5% reduction in static instruction count. Prior to compression, 11% of function calls displaced more than 1 MiB, exceeding the reach of the JAL instruction and requiring a two-instruction sequence. After compression, the total kernel text size became less than 1 MiB, eliminating all multi-instruction call sequences.

Table 5.7 dissects this data differently, showing the contribution to compression ratio of each RVC instruction, sorted by their frequency in the SPEC benchmarks. Unsurprisingly, data movement instructions dominate the static code size savings, owing primarily to their roles in function calls, prologues, and epilogues.

More importantly, the per-instruction breakdown highlights the importance of not overfitting the ISA design to one particular class of program. For example, had we only considered the SPEC benchmarks, we would have quickly concluded that instructions mostly used for bit-twiddling, like C.SRLI, C.SRAI, and C.ANDI, weren't good candidates for inclusion: collectively, they only contribute 0.22% to the compression ratio for those programs. CoreMark, though, emphasizes small integers and so uses these instructions prolifically, deriving 1.71% of the savings from them. Similarly, the Linux kernel would suggest that including the 32-bit stack-pointer-relative loads and stores (C.LWSP and C.SWSP) wouldn't have been helpful for 64-bit programs, but SPEC contradicts that observation.

While the compression ratio over the base ISA is one figure of merit, comparing absolute code size to other ISAs provides a more useful ground truth. After all, the more loosely encoded the base ISA, the more compressible it is. Figure 5.8 shows the static code size of the SPEC benchmarks for several 32-bit and 64-bit ISAs, normalized to RV32C and RV64C, respectively. GCC 5.2 was used for all experiments, at the `-O2` optimization level but with code alignment disabled. RV32C is considerably denser than IA-32, ARMv7, MIPS, and microMIPS code, and roughly ties ARM Thumb-2. Similarly, RV64C is quite a bit denser than x86-64, ARMv8, MIPS64, and microMIPS64. Table 5.6 presents the data for each benchmark individually.

The data lead to two interesting observations. First, the lone variable-length CISC ISA in the roundup, x86, has surprisingly poor code density. IA-32 code is only modestly smaller than RV32, and far larger than RV32C. This inefficiency is due in part to an archaic calling convention that mandates arguments be passed on the stack, not in registers, but another explanation is more fundamental: its architects have not always exercised care in managing its opcode space. One might reasonably expect, for example, that the precious 1-byte encoding space would encode only the most ubiquitous operations. And one would be incorrect: in a particularly galling example, six of the 256 patterns are dedicated to instructions that manipulate binary-coded decimal quantities, an operation so rare that GCC does not even deign to emit them. While x86-64 repented for that sin by deprecating those instructions, it is nevertheless even less dense, owing in large part to the extra 1-byte prefix necessary to encode the use of the expanded architectural register set.

Second, ARM's variable-length encoding, Thumb-2, is quite a bit denser than the ARMv7-A base ISA and performs similarly to RV32C in this comparison. Yet, ARM has decided not to include variable-length instructions in its 64-bit ISA, ARMv8. Although the latter offers good code size for a fixed-length encoding—it is about 8% denser than RV64 code,

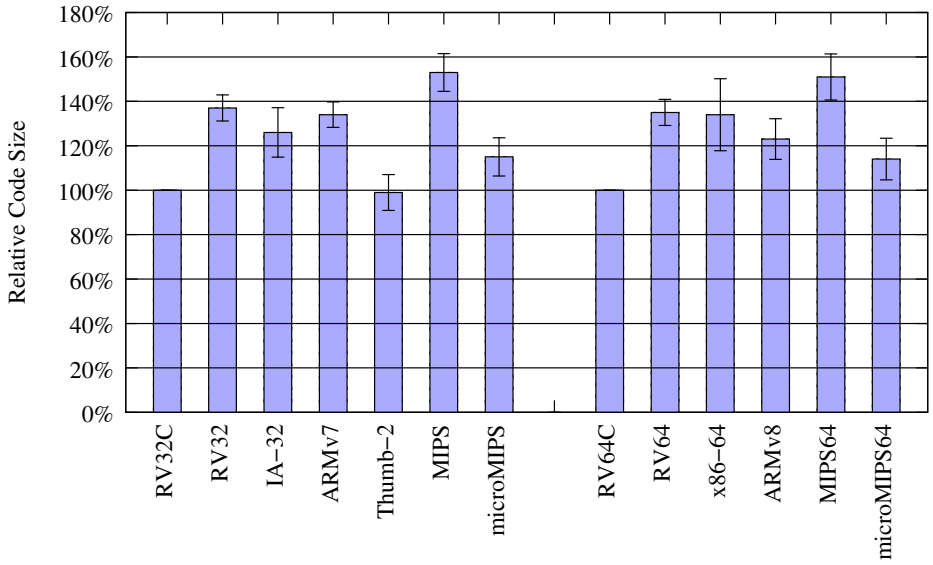

Figure 5.8: SPEC CPU2006 code size for several ISAs, normalized to RV32C for the 32-bit ISAs and RV64C for the 64-bit ones. Error bars represent ±1 standard deviation in normalized code size across the 29 benchmarks.

owing in large part to its load-pair and store-pair instructions—it cannot compete in code size with a variable-length encoding. Accordingly, ARM's first high-performance ARMv8 implementation, the Cortex-A57 [14], has a 50% larger instruction cache than its ARMv7 predecessor, the Cortex-A15 [13].

Dynamic Code Compression

The average savings in dynamic instruction fetch traffic closely reflects the static code size savings. RVC CoreMark and Dhrystone fetch 29.2% and 29.3% fewer instruction bytes than their RISC-V counterparts. In booting the Linux kernel there is a 26.1% reduction. On the SPEC reference inputs, SPECint sees a 26.9% savings and SPECfp saves 22.4%. From benchmark to benchmark, however, there is considerably more variation in dynamic code compression than in static code compression, as Figure 5.9 shows. This effect is due primarily to the dynamic dominance of a small sample of static code in several of the programs, but it is often an artifact of arbitrary compiler behavior. A single unlucky code generation decision might render a hot loop entirely incompressible. The difference in overall static code size might barely register, while the instruction fetch traffic would increase dramatically.

A representative example of this phenomenon appeared in `libquantum`, an implementation of Shor's algorithm, which spends about half of the execution time in a short loop evaluating a Toffoli quantum gate. An arbitrary register allocation decision, shown in Figure 5.10, resulted in $\frac{1}{3}$ of the instructions needlessly being incompressible, dragging the

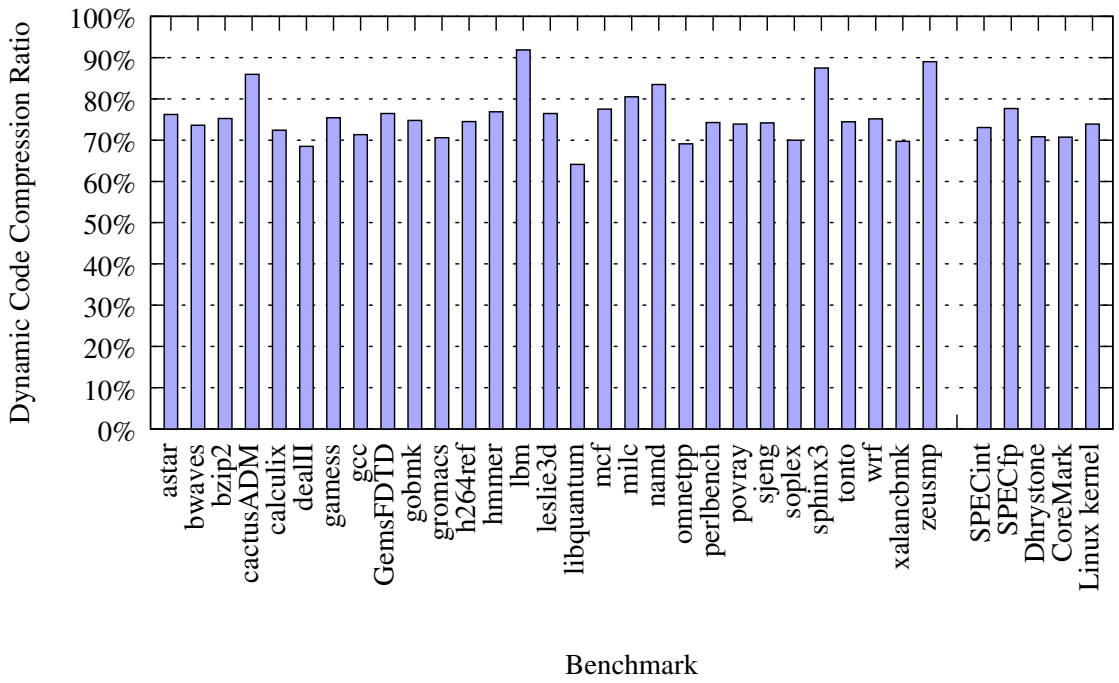

Figure 5.9: Dynamic compression of RVC code compared to RISC-V code in the SPEC CPU2006 benchmark suite, Dhrystone, CoreMark, and the Linux kernel.

possible dynamic savings from 35% down to 27%. Adjusting the compiler's cost model to penalize the use of non-RVC registers in code presumed to be hot worked around the problem at no discernible cost. Obviously, profiling data would improve this technique, since otherwise the static compiler must hazard a guess as to which code is likely to be frequently executed.

Of course, some of the variation in compressibility is fundamental and cannot be addressed with compiler tweaks. `lbm`, a computational fluid dynamics program, provides a case in point. At 8.1% savings, it has the least dynamic compression of any benchmark we examined. Nearly all of the runtime is spent in a single large loop that consists primarily of floating-point computational instructions, and uses 27 integer registers and 29 floating-point registers. The relative lack of locality of register reference makes this code difficult to compress, even if we had designed RVC to support a broader range of floating-point instructions.

Table 5.8 breaks down the dynamic fetch traffic savings on a per-instruction basis. Given the substantial variation of the dynamic compressibility between benchmarks, it is unsurprising that there is little consensus on the popularity of the various RVC instructions at runtime. For example, C.FLD, which barely registers in static code size reduction, is the fourth-most frequently executed RVC instruction in all of SPEC—even including SPECint. Meanwhile, it is unused in Dhrystone, CoreMark, and Linux. Similarly, bit-twiddling instructions are

```
00: 0104b883 ld      a7,16(s1)        00: 6898     c.ld    a4,16(s1)
04: 2505     c.addiw a0,1             02: 2505     c.addiw a0,1
06: 98be     c.add   a7,a5            04: 9742     c.add   a4,a6
08: 0088b803 ld      a6,8(a7)         06: 671c     c.ld    a5,8(a4)
0c: 07c1     c.addi  a5,16            08: 0841     c.addi  a6,16
0e: 00c846b3 xor     a3,a6,a2         0a: 00c7c6b3 xor     a3,a5,a2
12: 00b87833 and     a6,a6,a1         0e: 8fed     c.and   a5,a1
16: 00b81563 bne     a6,a1,20         10: 00b79563 bne     a5,a1,1a
1a: fee543e3 blt     a0,a4,0          14: ff1546e3 blt     a0,a7,0

             BEFORE                                AFTER
```

Figure 5.10: Code snippet from `libquantum`, before and after adjusting the C compiler's cost model to favor RVC registers in hot code. The compiler tweak reduced the size of the code from 30 to 24 bytes.

ubiquitous in CoreMark, despite being essentially unused in the other benchmarks.

Performance Implications

One of our main goals in defining RVC is to improve energy efficiency by means of reducing execution time. Generally, RVC code should not reduce performance as compared to RISC-V code. Misalignment of branch targets can reduce frontend performance, however, by inducing an extra pipeline bubble on taken branches. Several microarchitectural techniques exist to mitigate this performance loss and are regularly employed by superscalar processors, even those with fixed-width instructions. Among them are frontend decoupling, which allows frontend stalls to overlap backend stalls and is effective when combined with accurate branch prediction, and instruction cache banking, which can eliminate the problem almost entirely. Alternatively, the performance loss can be addressed entirely in software by re-aligning branch targets. To do so indiscriminately would increase code size considerably—about 3% on SPEC—but profiling feedback could be used to align only those targets of branches that are dynamically frequent and usually taken. In the absence of dynamic information, simply aligning loop bodies is a reasonable static heuristic: loop heads are typically the most common targets of taken branches, and they are statically less common than other branches.

More commonly, RVC will improve performance as compared to RISC-V code, by means of reducing cache misses, TLB misses, and page faults. To demonstrate this point, I simulated the SPEC CPU2006 benchmarks on idealized RISC-V and RVC processors that execute one instruction per cycle, except for cache misses, which block for 50 cycles. The processors all have 16 KiB 2-way set-associative data caches with 32-byte cache lines and a write-allocate,

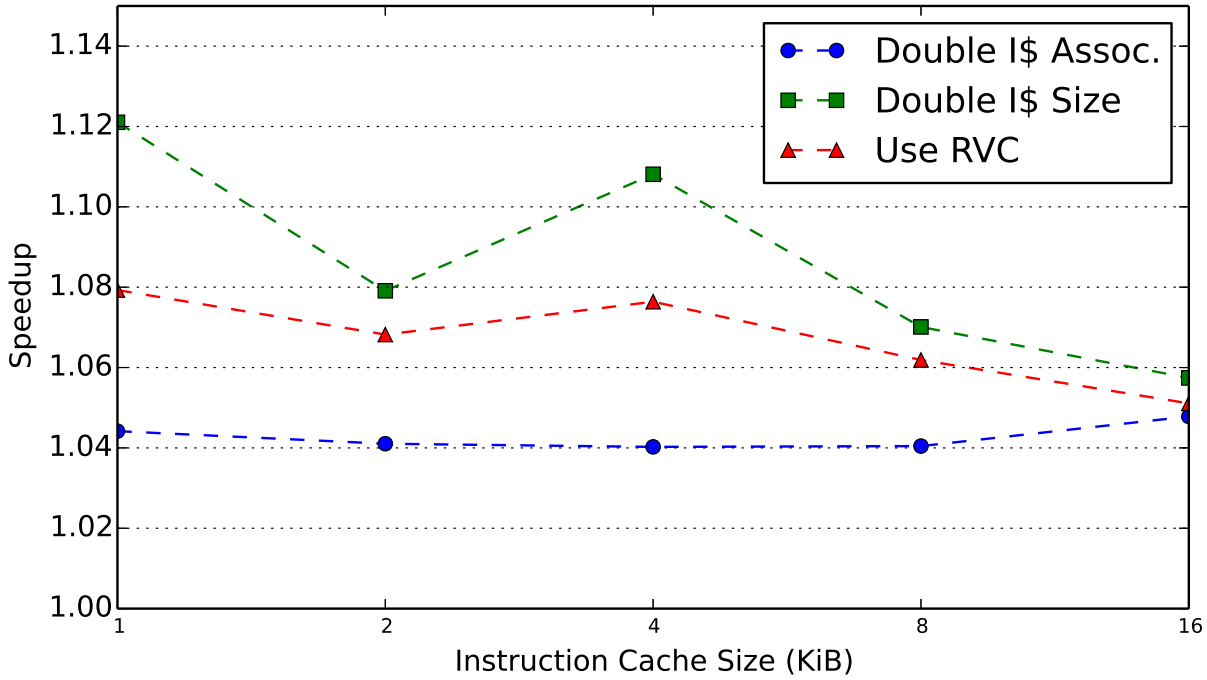

Figure 5.11: Speedup of larger caches, associative caches, and RVC over a direct-mapped cache baseline, for a range of instruction cache sizes.

write-back policy. They differ in their instruction cache configurations, which range from 1 KiB direct-mapped to 16 KiB 2-way set-associative.

Using the RISC-V processors with direct-mapped caches as baselines, Figure 5.11 shows the speedups that result from three changes: doubling the instruction cache size, making the instruction cache two-way set-associative, or running the RVC version of the same program. Across the range of caches, using RVC instead of RISC-V is more beneficial than increasing the associativity, and almost as performant as doubling the cache size.

Since the cost of implementing RVC is small—700 gates in one implementation [90], roughly the marginal area of 256 bytes of SRAM [96]—RVC obtains these performance benefits at substantially less area cost than increasing the instruction memory size. Alternatively, RVC can be used to reduce the area and power of a RISC-V processor without reducing its performance. Although a more detailed study is necessary to quantify the effect on energy efficiency, we note that instruction cache access contributes significantly to overall energy dissipation (27% in one study [69]) and that access energy increases rapidly with size and associativity. Hence, by enabling the use of less aggressive instruction caches without reducing performance, RVC has great potential to improve the energy efficiency of RISC-V processors.

5.6 The Load-Multiple and Store-Multiple Instructions

Arguably the most important job of an instruction-set architect is deciding what features should be left out. In defining RVC, the most difficult choice we made was whether to include *load-multiple* and *store-multiple* instructions, which load or store consecutive words in memory to or from a block of registers. Other compressed RISC ISAs, like microMIPS and Thumb, have included these instructions, primarily to reduce callee-saved register spill and fill code at procedure entry and exit. By our measurements, these can result in considerable code size savings: for example, the RVC Linux kernel text would become about 8% smaller with the use of these instructions in function prologues and epilogues.

Yet, after careful consideration, we opted against including them in RVC. The main reason was that they would violate our design constraint that each RVC instruction expand into a RISC-V instruction, thereby requiring compilers be RVC-aware and complicating processor implementations. They are also likely to be implemented with low performance in some superscalar microarchitectures, which would prohibit the issue of other instructions during the multi-cycle execution of the load- or store-multiple. Further reducing performance, especially for statically scheduled microarchitectures, is the inability of the compiler to co-schedule prologue and epilogue code with other code in the body of the function. Finally, in machines with virtual memory, these instructions can trigger page faults in the middle of their execution, complicating the implementation of precise exceptions or requiring a new restart mechanism.

The final nail in their coffin came with the realization that, when static code size matters more than runtime performance, we can obtain most of the benefit of load-multiple and store-multiple with a purely software technique. Since prologue and epilogue code is generally the same between functions, modulo the number of registers saved or restored, we can factor out this code into shared prologue and epilogue *millicode* routines[11]. These routines must have an alternate calling convention, since the link register must be preserved during their execution. Fortunately, unlike ARM and MIPS, RISC-V's jump-and-link instruction can write the return address to any integer register, rather than clobbering the ABI-designated link register. Other than that distinction, these millicode routines behave like ordinary procedures.

Figure 5.12 shows this technique in action, using as an example a naïve recursive implementation of the factorial function. The instructions that spill `ra` and `s0` to the stack are replaced with a call to `prologue_2`, and the instructions that reload them are replaced with a tail call to `epilogue_2`. In this case, factoring out the common prologue and epilogue code reduce code size by 19%, not counting the size of the shared routines. Figure 5.13 provides sample implementations of the two millicode routines, `prologue_2` and `epilogue_2`.

[11] Millicode is a technique pioneered by IBM in their S/390 architecture [70], in which complex instructions are implemented by high-level microcode routines that are in turn implemented with low-level microcode. RISC-V millicode is analogous, but the implementations take the form of normal RISC-V instructions.

```
uint64_t factorial(uint64_t x) {
  if (x > 0)
    return factorial(x - 1) * x;
  return 1;
}
```

```
00: cd11     c.beqz a0, 1c
02: 1141     c.addi sp, sp, -16
04: e406     c.sdsp ra, 8(sp)
06: e022     c.sdsp s0, 0(sp)
08: 842a     c.mv   s0, a0
0a: 157d     c.addi a0, -1
0c: ff5ff0ef jal    ra, factorial
10: 02850533 mul    a0, a0, s0
14: 60a2     c.ldsp ra, 8(sp)
16: 6402     c.ldsp s0, 0(sp)
18: 0141     c.addi sp, 16
1a: 8082     c.jr   ra
1c: 4505     c.li   a0, 1
1e: 8082     c.jr   ra
```

WITHOUT COMPRESSED PROLOGUES/EPILOGUES

```
00: c919     c.beqz a0, 16
02: 016002ef jal    t0, prologue_2
06: 842a     c.mv   s0, a0
08: 157d     c.addi a0, -1
0a: ff7ff0ef jal    ra, factorial
0e: 02850533 mul    a0, a0, s0
12: 0100006f jal    x0, epilogue_2
16: 4505     c.li   a0, 1
18: 8082     c.jr   ra
```

WITH COMPRESSED PROLOGUES/EPILOGUES

Figure 5.12: Naïve method to compute the factorial of an integer, both without and with prologue and epilogue millicode calls.

```
prologue_2:
    00: 1141     c.addi sp, -16
    02: e406     c.sdsp ra, 8(sp)
    04: e022     c.sdsp s0, 0(sp)
    06: 8282     c.jr   t0
```

```
epilogue_2:
    00: 60a2     c.ldsp ra, 8(sp)
    02: 6402     c.ldsp s0, 0(sp)
    04: 0141     c.addi sp, 16
    06: 8082     c.jr ra
```

Figure 5.13: Sample implementations of prologue and epilogue millicode routines for saving and restoring `ra` and `s0`.

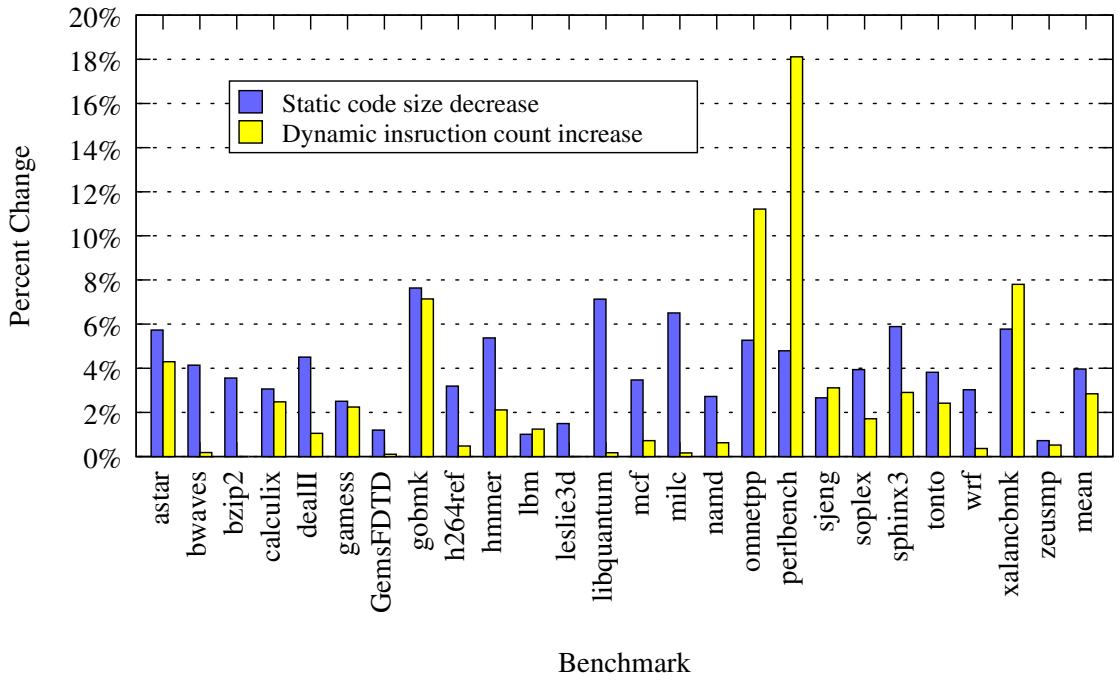

Figure 5.14: Impact on static code size and dynamic instruction count of compressed function prologue and epilogue millicode routines.

Unsurprisingly, there is a code size/performance tradeoff in the implementation of these routines. To minimize dynamic instruction count, one could have as many prologue and epilogue routines as there are callee-saved registers—13 of each in the standard calling convention. With this approach, prologues take two more dynamic instructions than if they had been inlined; epilogues take just one more, since they are implemented using a tail call. These routines would be fairly large, though, taking a collective 468 bytes in RV64C. At the other extreme is a single set of routines that can handle any register count, at the cost of more dynamic instructions per invocation. We opted for a hybrid approach with dedicated routines only for the common case of saving or restoring at most two registers; functions that save more registers must call the slower routine. This approach takes 122 bytes of code in RV64C.

To evaluate this approach to prologue and epilogue compression, we implemented it in our GCC port and ran a subset of the SPEC CPU2006 benchmarks. (We had to exclude those that throw C++ exceptions, because our implementation does not yet furnish the metadata needed to catch the exceptions.) Figure 5.14 shows the static code size savings of each benchmark over the plain RVC version and the dynamic instruction count *increase* for same. The improvement in code size is appreciable—an average of 4% savings—but comes with a 3% average increase in dynamic instruction count. The variance in the latter metric is significant, though; some benchmarks perform about the same, whereas those dominated

```
00: 02b7a023 sw x11, 32(x15)          02: 202302b7 lui  x5, 0x20230
04: 82822023 sw x8, -2016(x4)         06: 8282     c.jr x5
```

Figure 5.15: Interpretation of eight bytes of RVC code, depending on whether the code is entered at byte 0 or at byte 2.

by very frequent function calls, especially `omnetpp` and `perlbench`, become markedly slower. In the latter case, profiling feedback could be used to compress all but the most commonly called functions, obtaining the bulk of the savings without the corresponding performance loss.

We also evaluated the effectiveness of this technique on the RVC Linux kernel, which comprises many short functions and so should be a good candidate for this technique. Indeed, the kernel's text shrank by an impressive 7.5%, while the number of instructions executed during kernel boot increased by just 2.1%. This result suggests that, when employed judiciously, sharing the common prologue and epilogue code can provide a good tradeoff between code size and runtime performance.

It is interesting to contrast this technique with the register windows of the RISC-I and SPARC architectures. In both cases, the code to save and restore the callee-saved registers has been factored out, making function prologues and epilogues very compact. But the similarities end there. In the windowed architectures, the common save and restore code resides in the operating system kernel and is invoked via synchronous exception on register window overflow or underflow. This process is painfully slow, and it is exacerbated by the lack of information available to the lazy save and restore routines: they know nothing of the register usage pattern and so must conservatively spill or fill the entire window. The result is very fast calls and returns when the call stack is no deeper than the window size, and very slow ones otherwise. In contrast, these compressed prologue/epilogue routines do nothing to improve performance beyond reducing instruction cache misses; on the other hand, they do not have the performance cliffs of the register-window approach.

5.7 Security Implications

One feature of the base RISC-V ISA encoding is that instructions are all four bytes long and must be aligned to a four-byte boundary. Attempts to jump to the middle of an instruction word generate an exception. RVC, by design, lacks this property; indeed, the semantics of jumping into the middle of a four-byte instruction are well defined. Consider the RVC code in Figure 5.15, which has been disassembled twice, first starting at byte 0 and second starting at byte 2. If execution begins at byte 0, the code performs two stores to memory. If execution begins at byte 2, the code instead transfers control to address `0x20230000`.

This property of RVC complicates the task of software fault isolation tools (a.k.a. sandboxes), which aim to prove the safety of untrusted code, or rewrite it so that it is provably

safe. The dual interpretations of RVC code require either that both paths be proven safe, or that a higher-level constraint prevents the second path from executing. Fortunately, the latter approach has been proven to be possible for the more general problem of the variable-length x86 [64], and the strategy has been successfully deployed in the Google Native Client [107]. The key idea is to notionally divide instruction memory into 2^n-byte chunks, then constrain code generation so that basic blocks begin only on chunk boundaries and that no instructions span these chunks. To guarantee that indirect jumps cannot enter the middle of a chunk, the n least-significant bits of their targets must first be masked off (e.g., with a C.ANDI instruction). Finally, to guarantee that this masking instruction cannot itself be skipped, it must reside in the same chunk as the indirect jump.

Static analysis can easily verify in linear time that an RVC program satisfies these properties, so the sandboxing problem is not made substantially more difficult by the variable-length encoding.

5.8 Discussion

RISC-V, like its RISC predecessors, is not a particularly densely encoded instruction set architecture. Its regular, fixed-width encoding simplifies pipelined microarchitectures, helping to make the ISA suitable for research and educational purposes, but this property also serves as an Achilles' heel in domains where code size is a major concern. Embedded systems are a well-known example, as their cost and form factor strongly depend on memory size, but not the only one. Applications processors in mobile devices account for a large fraction of the power budget; a loose ISA encoding necessitates a large, leaky instruction cache, exacerbating the problem. Commercial workloads in large servers often have massive instruction working sets and are sometimes bound by the performance of the instruction memory system. The RVC extension increases RISC-V's applicability to all of these domains by improving code density by 25%–30%, resulting in smaller programs than all of the commercially popular 64-bit instruction sets.

Curiously, the two most popular general-purpose instruction sets are trending in the opposite direction. Intel's x86 never offered especially compact code, despite its variable-length encoding, but AMD's 64-bit extension, x86-64, exacerbated this problem. The backwards-incompatible change to wider addresses provided an opportunity to entirely recode the ISA, but the instruction set architects instead chose to keep the ISAs as similar as possible, presumably to minimize instruction decoding cost. (Ironically, the x86 ISA is so complex that adding a second, parallel instruction decoder would have barely added to the cost.) Indeed, many x86-64 instructions are larger than their IA-32 counterparts. While AMD's architects wisely doubled the depth of the anemic integer register set, they also added a one-byte penalty to use the new registers—even though most instructions only need one or two additional bits of register addresses. New AVX instructions using Intel's amusingly-named VEX prefix are as long as 11 bytes [47].

ARM's decision not to bring their Thumb extension into the ARMv8 ecosystem is more

perplexing still, given the ubiquity of their processors in deeply embedded systems. The larger instruction caches in their newest microprocessor offerings indicate that their microarchitects are keenly aware of the code size deficiency of their new instruction set. Why they traded a relatively small amount of logic for a relatively large amount of SRAM remains a mystery to us.

Where 32-bit addresses suffice, RVC is competitive in code size with ARM's Thumb-2 extension and superior to Intel's IA-32. In domains that demand both 64-bit addresses and dense code, RISC-V with the RVC extension will prove to be superior to both x86-64 and ARMv8. We believe that RVC will be a popular RISC-V instruction set extension for simple resource-constrained implementations and aggressive high-performance ones alike.

Instruction	Format	Base Instruction	Meaning
C.ADD	CR	`add rd, rd, rs2`	Add registers
C.ADDI	CI	`addi rd, rd, imm[5:0]`	Increment register
C.ADDI16SP	CI	`addi x2, x2, imm[9:4]`	Adjust stack pointer
C.ADDI4SPN	CIW	`addi rd′, x2, imm[9:2]`	Compute address of stack variable
C.ADDIW	CI	`addiw rd, rd, imm[5:0]`	Increment 32-bit register (RV64)
C.ADDW	CR	`addw rd, rd, rs2`	Add 32-bit registers (RV64)
C.AND	CS	`and rd′, rd′, rs2′`	Bitwise AND registers
C.ANDI	CI	`andi rd′, rd′, imm[5:0]`	Bitwise AND immediate
C.BEQZ	CB	`beq rs1′, x0, offset[8:1]`	Branch if zero
C.BNEZ	CB	`bne rs1′, x0, offset[8:1]`	Branch if nonzero
C.EBREAK	CR	`ebreak`	Breakpoint
C.FLD	CL	`fld rd′, offset[7:3](rs1′)`	Load double float
C.FLDSP	CI	`fld rd, offset[8:3](x2)`	Load double float, stack
C.FLW	CL	`flw rd′, offset[6:2](rs1′)`	Load single float (RV32)
C.FLWSP	CI	`flw rd, offset[7:2](x2)`	Load single float, stack (RV32)
C.FSD	CL	`fsd rs2′, offset[7:3](rs1′)`	Store double float
C.FSDSP	CSS	`fsd rs2, offset[8:3](x2)`	Store double float to stack
C.FSW	CL	`fsw rs2′, offset[6:2](rs1′)`	Store single float (RV32)
C.FSWSP	CSS	`fsw rs2, offset[7:2](x2)`	Store single float to stack (RV32)
C.J	CJ	`jal x0, offset[11:1]`	Jump
C.JAL	CJ	`jal x1, offset[11:1]`	Jump and link (RV32)
C.JALR	CR	`jalr x1, rs1, 0`	Jump and link register
C.JR	CR	`jalr x0, rs1, 0`	Jump register
C.LD	CL	`ld rd′, offset[7:3](rs1′)`	Load double-word (RV64)
C.LDSP	CI	`ld rd, offset[8:3](x2)`	Load double-word, stack (RV64)
C.LI	CI	`addi rd, x0, imm[5:0]`	Load immediate
C.LUI	CI	`lui rd, imm[17:12]`	Load upper immediate
C.LW	CL	`lw rd′, offset[6:2](rs1′)`	Load word
C.LWSP	CI	`lw rd, offset[7:2](x2)`	Load word, stack
C.MV	CR	`add rd, x0, rs2`	Copy register
C.OR	CS	`or rd′, rd′, rs2′`	Bitwise OR registers
C.SD	CL	`sd rs2′, offset[7:3](rs1′)`	Store double-word (RV64)
C.SDSP	CSS	`sd rs2, offset[8:3](x2)`	Store double-word to stack (RV64)
C.SLLI	CI	`slli rd, rd, imm[5:0]`	Shift left, immediate
C.SRAI	CB	`srai rd′, rd′, imm[5:0]`	Arithmetic shift right, immediate
C.SRLI	CB	`srli rd′, rd′, imm[5:0]`	Logical shift right, immediate
C.SUB	CS	`sub rd′, rd′, rs2′`	Subtract registers
C.SUBW	CS	`subw rd′, rd′, rs2′`	Subtract 32-bit registers (RV64)
C.SW	CL	`sw rs2′, offset[6:2](rs1′)`	Store word
C.SWSP	CSS	`sw rs2, offset[7:2](x2)`	Store word to stack
C.XOR	CS	`xor rd′, rd′, rs2′`	Bitwise XOR registers

Table 5.3: RV32C and RV64C instruction listing.

15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0	
000	nzimm[5:4|9:6|2|3]				rd′	00	C.ADDI4SPN
001	imm[5:3]		rs1′	imm[7:6]	rd′	00	C.FLD
010	imm[5:3]		rs1′	imm[2|6]	rd′	00	C.LW
011	imm[5:3]		rs1′	imm[7:6]	rd′	00	C.LD
101	imm[5:3]		rs1′	imm[7:6]	rs2′	00	C.FSD
110	imm[5:3]		rs1′	imm[2|6]	rs2′	00	C.SW
111	imm[5:3]		rs1′	imm[7:6]	rs2′	00	C.SD
000	0	0		0		01	C.NOP
000	nzimm[5]	rs1/rd≠0		nzimm[4:0]		01	C.ADDI
001	imm[5]	rs1/rd≠0		imm[4:0]		01	C.ADDIW
010	imm[5]	rs1/rd≠0		imm[4:0]		01	C.LI
011	nzimm[9]	2		nzimm[4|6|8:7|5]		01	C.ADDI16SP
011	nzimm[17]	rs1/rd≠{0, 2}		nzimm[16:12]		01	C.LUI
100	nzimm[5]	00	rs1′/rd′	nzimm[4:0]		01	C.SRLI
100	nzimm[5]	01	rs1′/rd′	nzimm[4:0]		01	C.SRAI
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
100	1	11	rs1′/rd′	01	rs2′	01	C.ADDW
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
000	nzimm[5]	rd≠0		nzimm[4:0]		10	C.SLLI
001	imm[5]	rd		imm[4:3|8:6]		10	C.FLDSP
010	imm[5]	rd≠0		imm[4:2|7:6]		10	C.LWSP
011	imm[5]	rd≠0		imm[4:3|8:6]		10	C.LDSP
100	0	rs1≠0		0		10	C.JR
100	0	rd≠0		rs2≠0		10	C.MV
100	1	0		0		10	C.EBREAK
100	1	rs1≠0		0		10	C.JALR
100	1	rd≠0		rs2≠0		10	C.ADD
101	imm[5:3|8:6]			rs2		10	C.FSDSP
110	imm[5:2|7:6]			rs2		10	C.SWSP
111	imm[5:3|8:6]			rs2		10	C.SDSP

Table 5.4: RV64C instruction encoding.

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	0			00	*Illegal Instruction*
100	—			00	*Reserved for future use*
000	0	≠0	0	01	*Reserved hint*
001	—	0	—	01	*Reserved for future use*
010	—	0	—	01	*Reserved hint*
011	0	0	≠0	01	*Reserved hint*
011	1	0	—	01	*Reserved hint*
011	0	—	0	01	*Reserved for future use*
100	0	—	0	01	*Reserved hint*
100	111	—	1 —	01	*Reserved for future use*
000	0		≠0	10	*Reserved hint*
011	—	0	—	10	*Reserved for future use*
100	—	0	≠0	10	*Reserved hint*

Table 5.5: Reserved encodings in RV64C.

	32-bit ISAs						64-bit ISAs				
Benchmark	RV	x86	ARM	Thm2	MIPS	μM	RV	x86	ARM	MIPS	μM
astar	1.41	1.30	1.26	0.94	1.53	1.06	1.37	1.27	1.26	1.57	1.07
bwaves	1.42	1.32	1.29	0.94	1.45	1.22	1.32	1.69	1.19	1.45	1.20
bzip2	1.44	1.15	1.35	0.94	1.53	1.08	1.42	1.11	1.23	1.55	1.14
cactusADM	1.34	1.29	1.35	1.02	1.57	1.18	1.34	1.36	1.28	1.56	1.16
calculix	1.38	1.34	1.36	1.01	1.52	1.17	1.38	1.43	1.21	1.42	1.05
dealII	1.45	1.35	1.33	0.93	1.59	1.09	1.45	1.38	1.27	1.60	1.06
gamess	1.34	1.22	1.29	0.96	1.39	1.14	1.31	1.34	1.17	1.34	1.11
gcc	1.35	1.16	1.39	0.95	1.46	1.10	1.34	1.19	1.24	1.48	1.15
GemsFDTD	1.31	1.41	1.32	0.97	1.48	1.24	1.31	1.60	1.17	1.59	1.39
gobmk	1.41	1.09	1.28	0.92	1.57	1.02	1.41	1.11	1.30	1.62	1.06
gromacs	1.37	1.20	1.27	1.00	1.46	1.14	1.28	1.34	1.13	1.42	1.10
h264ref	1.37	1.22	1.33	0.98	1.48	1.14	1.35	1.23	1.17	1.48	1.18
hmmer	1.38	1.26	1.32	0.95	1.55	1.07	1.37	1.29	1.26	1.54	1.01
lbm	1.20	0.99	1.41	1.24	1.61	1.38	1.19	1.23	1.06	1.43	1.26
leslie3d	1.41	1.39	1.38	0.97	1.47	1.19	1.36	1.67	1.24	1.47	1.22
libquantum	1.44	1.26	1.51	1.21	1.59	1.20	1.40	1.33	1.34	1.63	1.26
mcf	1.35	1.26	1.36	0.99	1.65	1.14	1.34	1.34	1.36	1.72	1.21
milc	1.35	1.33	1.41	1.07	1.67	1.24	1.37	1.42	1.34	1.62	1.19
namd	1.30	1.45	1.28	1.05	1.47	1.29	1.28	1.61	1.12	1.36	1.17
omnetpp	1.40	1.29	1.29	0.94	1.55	1.06	1.39	1.26	1.36	1.42	1.01
perlbench	1.36	1.25	1.45	1.00	1.61	1.12	1.35	1.28	1.31	1.60	1.09
povray	1.29	1.32	1.32	1.07	1.57	1.25	1.28	1.46	1.19	1.57	1.29
sjeng	1.30	1.12	1.29	0.96	1.43	1.08	1.32	1.11	1.19	1.44	1.06
soplex	1.41	1.40	1.38	1.03	1.69	1.22	1.40	1.40	1.33	1.66	1.19
sphinx3	1.35	1.23	1.30	0.97	1.52	1.08	1.34	1.25	1.28	1.50	1.02
tonto	1.39	1.47	1.33	0.95	1.50	1.15	1.39	1.63	1.20	1.49	1.17
wrf	1.39	1.28	1.31	0.91	1.34	1.04	1.34	1.34	1.10	1.31	1.01
xalancbmk	1.50	1.29	1.36	0.87	1.64	1.02	1.49	1.26	1.40	1.64	1.07
zeusmp	1.35	1.10	1.38	1.06	1.43	1.21	1.30	1.24	1.05	1.41	1.17
Geo. Mean	1.37	1.26	1.34	0.99	1.53	1.15	1.35	1.34	1.23	1.51	1.14

Table 5.6: SPEC CPU2006 code size for several ISAs, normalized to RV32C for the 32-bit ISAs and RV64C for the 64-bit ones. Thm2 is short for ARM Thumb-2; μM is short for microMIPS.

Instruction	RV32GC			RV64GC		Max
	Dhry-stone	Core-Mark	SPEC 2006	SPEC 2006	Linux Kernel	
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 5.7: 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.

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 5.8: 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.

Chapter 6

A RISC-V Privileged Architecture

The preceding three chapters have described the RISC-V user-level ISA and have largely steered clear of system-level issues. This descriptive tack is not merely stylistic: it also reflects a technical decision to orthogonalize the RISC-V user ISA and privileged architecture. The reasons for this separation are multifold:

- *It allows the user ISA to be shared across a wide variety of systems.* Real-time embedded systems with a trusted code base demand significantly different features of the privileged architecture than do hypervised server systems with I/O virtualization. If the user and privileged architectures are separated, though, the same user ISA can be used for both, reducing software development costs.
- *It facilitates experimentation in privileged architectures.* Researchers can devise and implement novel memory translation and protection schemes, new security features, and alternative I/O mechanisms without having to rewrite application code.
- *It simplifies the implementation of full virtualization.* Exposing privileged features to unprivileged software adds complexity to hardware-assisted virtualization, and can make classical virtualization impossible[1].

Consequently, rather than defining *the* RISC-V privileged architecture, we propose a *reference* RISC-V privileged architecture, *RPA*, designed to support Unix-like operating systems with page-based virtual memory. RPA naturally supports classical virtualization and can be extended to support hardware-accelerated virtualization, akin to the IBM System/370 [75]. Additionally, the virtual memory facility can be replaced with a lower-cost base-and-bounds

[1] A famous example of the latter is the x86 FLAGS register, which holds both user-visible condition codes and privileged fields, such as the interrupt-enable flag. User-mode writes to the privileged portion of the FLAGS register are ignored. This silent failure foils classical virtualization, in which the guest operating system runs in user mode: guest attempts to disable interrupts, for example, cannot be intercepted and emulated by the host OS. VMware heroically worked around this limitation with an intricate dynamic binary translation scheme [23].

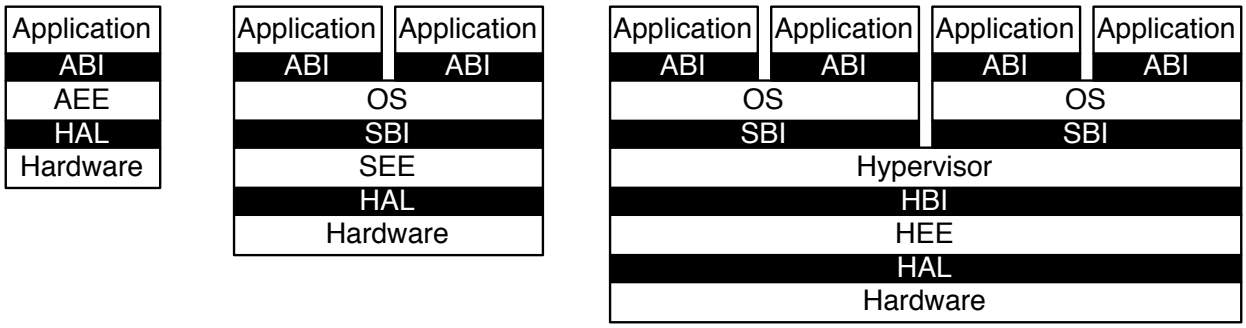

Figure 6.1: The same ABI can be implemented by many different privileged software stacks. For systems running on real RISC-V hardware, a hardware abstraction layer underpins the most privileged execution environment.

protection and translation scheme to support lighter-weight embedded systems. The remainder of this chapter describes RPA's major features, focusing on the interface to supervisor software. A more complete description is available in [103].

6.1 Privileged Software Interfaces

In a conventional general-purpose system, applications make requests of the operating system using a system-call convention defined in an application binary interface (ABI). Application code need not know details of the underlying mechanisms that implement the system call; it suffices to know the interface. This abstraction improves modularity. Indeed, user software may even be ignorant of the identity of its application execution environment (AEE): ordinarily, it is an operating system, but it could just as easily be an emulator that speaks the same ABI.

Curiously, most privileged execution environments do not afford operating system software the same courtesy. The OS typically performs such actions as arming timers and routing interrupts by interacting directly with the underlying hardware platform. This approach limits either implementation flexibility or OS portability, and it complicates and decelerates full virtualization. In contrast, RPA abstracts the supervisor execution environment (SEE) that underpins the operating system by way of a supervisor binary interface (SBI). The SEE may be a simple boot loader with primitive I/O abstraction, similar to the PC BIOS; or a hypervisor that fully virtualizes the I/O and memory systems; or even an SBI-level emulator. A uniform SBI allows the same OS code to run on any of these SEEs.

The analogy continues to the hypervisor, which interacts with the hypervisor execution environment (HEE) via a hypervisor binary interface (HBI). This design simplifies the implementation of recursive virtualization.

Of course, the abstraction must terminate at some point. For native RISC-V hardware systems, the lowest-level execution environment interacts with the hardware via a hardware

Level	Description
U	User/Application
S	Supervisor
H	Hypervisor
M	Machine/Trusted

Number of Levels	Supported Modes
1	M
2	M, U
3	M, S, U
4	M, H, S, U

Table 6.1: RPA privilege modes and supported privilege mode combinations.

abstraction layer (HAL). The HAL isolates the execution environment from the implementation details of the hardware platform, such as the location of control registers in the address map. Hiding platform-specific details improves the reusability of execution environment software. Figure 6.1 shows a logical view of this design.

6.2 Four Levels of Privilege

A secure system need only have two privilege modes—privileged, where operating system (OS) code runs, and unprivileged, in which application code executes. Such a scheme suffices to protect the system from errant user processes and protect user processes from each other; various ISAs, like SPARC, have successfully employed this strategy [87]. As long as all privileged instructions generate traps when executed in user mode, it also suffices to support *classical virtualization*, in which guest OSes systems run in unprivileged mode. In that scheme, the host OS, running in privileged mode, emulates privileged functionality on the guests' behalf.

Nevertheless, there are compelling reasons to provide additional privileged modes. The DEC Alpha, for example, provides a third, greater privilege level in which *PALcode* (Privileged Architecture Library) executes [77]. PALcode presents a high-level interface to OS code for certain low-level functionality, which, like RPA's SBI, abstracts the implementation details from the OS. The PALcode mechanism can also implement missing hardware functionality, allowing low-end implementations to omit hardware support for expensive operations.

Similarly, while a two-level scheme suffices to support virtualization, many guest OS operations can be significantly accelerated with additional hardware support. Minimizing the number of guest instructions that result in a trap into the hypervisor reduces virtualization overhead. The obvious means to reduce the frequency of these trapping events is to run the guest OS in privileged mode. Ordinarily, doing so would compromise the isolation between guests. To maintain inter-guest protection, we can add a hyperprivileged mode in which the hypervisor executes. Coupled with an additional memory protection scheme, the hypervisor can maintain isolation between virtual machines with much lower overhead than the classical virtualization approach.

Address Bits 11:10	Meaning
00	Read/Write
01	Read/Write
10	Read/Write
11	Read-Only

Address Bits 9:8	Minimum Privilege
00	U
01	S
10	H
11	M

Table 6.2: RPA privilege modes and supported privilege mode combinations.

Accordingly, RPA exposes four modes of increasing level of privilege. The least-privileged mode is *User* (U), in which application code normally executes. Above that is *Supervisor* (S) mode, which provides basic exception processing and virtual memory support; OS code normally executes in this level. The *Hypervisor* (H) mode is a placeholder for a privilege mode designed to host a virtual machine monitor; we have yet to define H-mode. Finally, the *Machine* (M) mode has unfettered access to all hardware features. The first three modes are optional; only M-mode is required. Table 6.1 summarizes the privilege modes and supported combinations. Providing only M-mode suffices for many embedded systems, though adding U-mode provides some isolation between system software and application code, easing debugging. An M/S/U system can support a traditional Unix-like OS. Adding H-mode further supports a hardware-assisted hypervisor.

6.3 A Unified Control Register Scheme

Each privilege mode requires a handful of control and status registers (CSRs) to support, among other features, exception processing and interrupt handling. To minimize both software and hardware complexity, all CSRs are accessed through the same six-instruction interface described in Chapter 3. In that chapter, the only CSRs defined were the read-only performance counters; the more-privileged modes add several read-only and writable CSRs.

The CSRs reside in an ample 12-bit address space, set by the immediate operand width in the CSR access instructions. In an effort to minimize the descriptive complexity of the privileged architecture and to simplify its hardware implementation, we divide the address space into privilege regions, as Table 6.2 shows. The two most-significant bits indicate whether a CSR is read-only, and the next two bits give the minimum privilege mode at which the register may be accessed. In a scheme described in [103], we further subdivide the address space into *standard* and *non-standard* subspaces, demarcating a region that non-standard extensions may use without fear of the space being reclaimed by future standard extensions.

Several CSRs, such as the user-level performance counters, must be writable at higher privilege levels. RPA addresses this need by allowing CSR *shadows*: a CSR may be redundantly mapped into the writable portion of a greater privilege mode's CSR address space. For

Name	Meaning
`sstatus`	Processor status register
`stvec`	Trap handler base address
`sie`	Interrupt-enable register
`sscratch`	Scratch register
`sepc`	Exception program counter
`scause`	Trap cause
`sbadaddr`	Bad address
`sip`	Interrupt-pending register
`sptbr`	Page table base register
`sasid`	Address-space identifier

Table 6.3: Listing of supervisor-mode control and status registers.

example, the user-level `cycle` counter has address `0xC00`, i.e., read-only with user privilege. Its writable shadow, `cyclew`, has address `0x900`, i.e., writable with supervisor privilege.

6.4 Supervisor Mode

The RPA supervisor mode provides an interface against which traditional Unix-like operating systems may be written. In keeping with the abstract SBI interface, only a minimal view of the machine state is exposed to supervisor software. In particular, the supervisor cannot directly interrogate the hardware to determine the existence of greater privilege levels.

The supervisor mode provides two main services to supervisor software: an exception processing facility and virtual memory management. RPA furnishes supervisor software with a handful of CSRs to avail itself of these features, which Table 6.3 lists. The most important of them is the `sstatus` register, which controls the privilege level, the global interrupt enable, and the status of the floating-point extensions and the non-standard extensions. In addition to controlling the availability of the extensions, the `sstatus` register indicates whether their architectural state has been modified, allowing the OS to avoid saving and restoring the state on context switches. The register's most-significant bit summarizes the dirtiness of the extension state, so that the OS can determine with a single BLT instruction whether it must save any of the state on a context switch.

Additional CSRs support exception processing. `stvec` holds the address to which the `pc` is set upon an exception. When an exception occurs, `sepc` is set to the address of the offending instruction. `scause` indicates why the exception occurred, and, if the cause was a misaligned or invalid address, `sbadaddr` records it.

To simplify virtualization and improve failure isolation, RPA favors an abstract I/O device model, even when the supervisor software is running bare-metal. In this model, device drivers execute in a separate process from the OS; the OS interacts with them via SBI calls. Communication with device drivers is thus similar to communication with other

processors. Accordingly, the RPA supervisor mode lacks device interrupts. In fact, there are only two interrupt classes: *software* interrupts, which are triggered by other threads of execution, and *timer* interrupts, which are triggered by the real-time counter. The `sip` CSR indicates if either class of interrupt is pending; corresponding bits in `sie` mask these interrupts. Since the mechanisms for timer and interprocessor interrupts vary from system to system, the SBI provides abstract interfaces to request them.

Virtual Memory

The RPA supervisor mode provides a page-based virtual memory system, in which both physical memory and the virtual address space are divided into fixed-size pages. A virtual page can map to any physical page, or none at all, as specified by an in-memory high-radix tree structure called the page table. To simplify memory allocation, each node in the tree is also the size of a page. The physical address of the root node is stored in the `sptbr` CSR, which can be swapped on context switches to give each process a unique virtual address space.

In both RV32 and RV64, pages are 4 KiB, as is the case for IA-32, x86-64, ARMv7, ARMv8, and SPARC V8. We had originally contemplated a larger page size, a position that proved to be remarkably contentious, in part because the page size is exposed to application software by way of the ABI. While portable software should not make assumptions about the page size—even for a given ISA—in practice, much software does. Choosing the most popular page size reduces the porting effort.

More importantly, the page size has a substantial performance impact for some applications. A greater page size increases the amount of memory that an address translation cache of a certain capacity can map. It also loosens the associativity constraints on virtually indexed, physically tagged caches[2], potentially reducing cache access energy. On the other hand, since pages are the memory allocation quantum, larger pages exacerbate internal fragmentation, thereby wasting physical memory. This phenomenon is perhaps most noticeable in an operating system's file cache: caching small files dramatically improves overall system performance, but sacrifices significant physical memory to internal fragmentation [97].

A mitigating factor in our decision to not increase the page size is that RPA supports *superpages* at any level of the page table structure. In RV32, the page table is a two-level radix-1024 tree, so in addition to 4 KiB pages, it also allows 4 MiB *megapages*. RV64 has multiple page table organizations, corresponding to different virtual address space sizes; the simplest 39-bit mode has a three-level radix-512 page table. This scheme gives not only 2 MiB megapages, but also 1 GiB *gigapages*[3]. While these superpages are difficult for application

[2] A virtually indexed, physically tagged cache is a design in which the cache is indexed in parallel with the address translation process. Of course, doing so requires the cache index to be known prior to address translation. Since the only part of the address known in advance is the page offset, the size of one cache set cannot exceed the page size.

[3] We gave the different superpage sizes idiomatic names to avoid confusion. We could have named them L1 and L2 superpages, but it would've been ambiguous whether we were counting from the root or the leaves.

			Supervisor			User		
Type	Meaning	Global	R	W	X	R	W	X
0	Pointer to next level of page table.							
1	Pointer to next level of page table—global mapping.	•			—			
2	Supervisor read-only, user read-execute page.		•			•		•
3	Supervisor read-write, user read-write-execute page.		•	•		•	•	•
4	Supervisor and user read-only page.		•			•		
5	Supervisor and user read-write page.		•	•		•	•	
6	Supervisor and user read-execute page.		•		•	•		•
7	Supervisor and user read-write-execute page.		•	•	•	•	•	•
8	Supervisor read-only page.		•					
9	Supervisor read-write page.		•	•				
10	Supervisor read-execute page.		•		•			
11	Supervisor read-write-execute page.		•	•	•			
12	Supervisor read-only page—global mapping.	•	•					
13	Supervisor read-write page—global mapping.	•	•	•				
14	Supervisor read-execute page—global mapping.	•	•		•			
15	Supervisor read-write-execute page—global mapping.	•	•	•	•			

Table 6.4: Page table entry types.

code to use directly, researchers at Rice University devised a scheme to automatically promote large, contiguous memory allocations to superpages [71]. *Transparent superpages* have since been implemented to great effect in Linux and FreeBSD, reducing the pressure on architects to provide larger base pages.

Page-based virtual memory systems provide not only address translation, but also a memory protection mechanism. Each virtual page can be configured with one of ten sets of permissions, encoded in a four-bit field in the page table entry. These correspond to types 2–11 in Table 6.4. Like most virtual memory systems, we allow pages to be marked as read-only or writable, either by the supervisor only or by the user as well. We additionally allow pages to be marked non-executable, which prevents a class of security attack, including buffer overflows that write instructions to the stack. Along the same lines, while the supervisor's permissions are ordinarily a superset of the user's permissions, we allow pages to be marked executable by the user but non-executable by the supervisor. This option prevents errant operating system code from inadvertently executing untrusted user code. In our Linux kernel port, every user-executable page is so marked.

Address-Translation Caching

A virtual-to-physical address translation must be performed for every instruction fetch and for every load and store. Since these translations require multiple memory accesses themselves, they would result in a slowdown of hundreds of percent if not somehow accelerated.

Hence, nearly all systems with virtual memory cache the results of address translations in a TLB[4]. To simplify the hardware, most systems do not keep these caches coherent with the page table; instead, they require that system software flush the translation caches upon page table modification. In fact, some architectures, like MIPS, even require that the OS fully manage the cache. In that design, TLB misses cause synchronous exceptions, and a trap handler refills the TLB.

The software-refill approach has two main advantages: it slightly reduces hardware cost, and it makes the hardware agnostic to the page table data structure. Indeed, with software refill, the operating system can even take an adaptive approach—for example, by switching from a radix tree to an inverted page table [25]. But it also has two significant drawbacks. First, it adds ISA complexity in other ways. The MIPS design, for example, requires additional instructions to refill and strike TLB entries. Second, and more importantly, it erects a performance bottleneck for high-performance systems. Software refill exceptions cause pipeline flushes, which are particularly expensive for dynamically scheduled microarchitectures. In contrast, with hardware TLB refill, these microarchitectures can often schedule useful work around the TLB miss, decoupling the execution units from the high-latency memory operation.

We felt the advantages of hardware TLB refill easily justified the minor cost of a hardware refill unit. The only remaining question was whether we could rationalize eliminating the possibility of alternative page table data structures, like inverted page tables. But researchers have shown that simple radix trees often perform better [50], and, when coupled with translation path caches, result in fewer DRAM accesses per TLB miss [19]. Accordingly, RPA specifies hardware TLB refill, using only the radix tree structure.

We briefly note that RPA does allow software TLB refill using M-mode support routines. Although we do not recommend this design, supervisor software would be none the wiser.

RPA does not require that TLBs be kept coherent with page table updates. Instead, it adds an instruction, SFENCE.VM, which guarantees that prior page table writes are ordered before subsequent address translation operations. This definition makes no mention of TLBs, but in TLB-based systems, it may indeed be implemented by flushing the TLB. The advantage of our approach is that it provides cleaner semantics with respect to the side effects of the flush operation, and that it supports a wider variety of address translation caching schemes. Its definition also avoids exposing a pipeline hazard, unlike many other architectures. For example, in the MIPS R4000, five instructions must be executed between writing a TLB entry and fetching an instruction from the corresponding virtual memory page [66].

[4]TLB stands for *Translation Lookaside Buffer*. It is an an anachronistic and esoteric name for what would be better described as an address translation cache.

6.5 Hypervisor Mode

RPA has reserved opcode space and CSR address space for a future hypervisor mode, but we have yet to define one. One of the most important functions of the hypervisor is to virtualize physical memory, and so the hypervisor mode will provide an additional layer of address translation, from *supervisor physical addresses* to *hypervisor physical addresses*. A plausible implementation would use an address translation scheme very similar to the one described in the previous section.

6.6 Machine Mode

Machine mode is the only required privilege mode in RPA, because it has access to all hardware features. Accordingly, M-mode software is assumed to be fully trusted. M-mode is designed to be sufficient to host simple embedded systems that derive little benefit from memory protection, but it also fills a crucial role in more complex designs. For systems with a supervisor-mode operating system, the primary role of M-mode is to abstract the hardware platform and provide any missing hardware features. In our RISC-V implementations, for example, M-mode software emulates misaligned loads and stores, unbeknownst even to supervisor software. It also emulates the standard floating-point extensions when they are unimplemented in hardware. In that respect, M-mode software is part and parcel of the hardware implementation.

M-mode does not, by default, translate memory addresses; it interacts with the physical memory system directly. It does, however, furnish a facility to execute loads and stores using a lower privilege mode's address translation scheme. This feature is especially useful for emulating missing hardware functionality.

An exception processing scheme very similar to the S-mode facility handles, by default, all traps, regardless of their source and ultimate destination. Trap-redirection instructions can quickly vector a subset of traps to less-privileged software when appropriate—for example, to the supervisor to handle virtual memory page faults. Interrupt handling is also similar to the S-mode approach, though most platforms will provide many more interrupt causes than are visible to S-mode, including non-maskable interrupts. M-mode provides a wait-for-interrupt instruction that may stall the processor until any interrupt becomes pending, saving energy. (It is defined in such a way that it can execute as a no-op in simple implementations.)

Device I/O mechanisms are implementation-defined, but in sophisticated systems, they are generally expected to use memory-mapped I/O. Simpler systems might use CSR-mapped I/O exclusively, which can be easier to implement because I/O operations may be identified upon decode, rather than after effective address generation. While it would be simpler for all systems to use CSR-mapped I/O, it is impractical because, for many applications, the I/O address space needs to be indexable. Additionally, some existing device drivers assume memory-mapped I/O and would have to be rewritten.

Interprocessor interrupts are effectively mandatory for multiprocessor systems. The mechanism is implementation-defined, but it is generally expected to follow the platform's device I/O conventions. In our implementations, all processors' CSRs are mapped into the physical address space, so processors can write each other's software interrupt pending bits directly.

Finally, a real-time counter and a comparator trigger timer interrupts. This lone mechanism provides timer interrupts for all less-privileged software; when multiple timers are required, they are multiplexed by M-mode software.

6.7 Discussion

The Reference Privilege Architecture provides a simple interface for traditional operating systems, hardware-accelerated hypervisors, and classical virtualization strategies. Its design is not yet complete—in particular, the hypervisor mode and the SBI have yet to be defined. We expect all but the hypervisor layer to be finalized in 2016.

A rich design space exists in privileged instruction set architecture, much of it yet to be explored. RPA's simplicity makes it suitable for some forms of experimentation, but more divergent approaches to computer security problems in particular may favor replacing large parts of the architecture. In particular, coupling memory protection to address translation, while expedient, is an anachronistic approach to a pivotal problem that fails to provide the flexibility and granularity that modern software demands. We excitedly await the fruits of computer architecture and operating systems researchers' labor on this and other important problems in the privileged architecture space.

Chapter 7

Future Directions

In defining RISC-V, we believe we have succeeded in making a broadly applicable user-level ISA. It is free, open, simple, and modular. Yet, much design work remains. Notably, the privileged architecture is incomplete: the reference hardware platform and the various binary interfaces remain to be specified. There is also much to be done in the user ISA.

For example, we see two outstanding issues in the multiprocessor memory system. The first is definitional: an operational model of the memory system remains to be specified. Informally, we lean towards a model that mandates store atomicity and respects read-after-write hazards in the instruction stream. We briefly note that such a stricture does not preclude techniques like value speculation [62]; it just increases their implementation complexity, placing greater onus on the misspeculation detection mechanisms.

The second is a missing feature: multi-word atomicity. The **A** extension provides atomics only on word-sized quantities. In contrast, some ISAs provide a double-word compare-and-swap (DW-CAS) operation. The most common use of DW-CAS is to sidestep the ABA problem[1] by using the second word as a sequence number. This workaround is unnecessary in RISC-V, since the load-reserved/store-conditional primitives do not suffer from the ABA problem. But the omission of DW-CAS still presents the potential for incompatibility. Nevertheless, we view DW-CAS as a band-aid, and instead favor a more general multi-word atomicity facility, along the lines of the original transactional memory proposals [41]. But it seems prudent to wait and see what direction the programming language community takes, rather than over-architecting a heavyweight ISA extension that might go largely unused.

Another pivotal ISA feature we have yet to specify in RISC-V is efficient support for data-level parallelism. The most popular ISAs provide a fixed-width SIMD architecture. This rigid approach exposes a forward-incompatible programming model: as successive ISA versions expand the SIMD width, old software cannot exploit the increased parallelism without recompilation. Likewise, software compiled for wider SIMD cannot execute natively on old hardware. In contrast, traditional vector architectures need not expose the vector length

[1] As described in Section 4.2, the ABA problem arises when a memory location is modified from the value A to the value B, then modified back to the value A. Since the success or failure of CAS is based upon value equality, CAS cannot detect this scenario.

statically [76]. The design is therefore scalable, in that the same program binaries execute at full performance on machines with any vector length. This vastly superior architectural paradigm will eventually form the RISC-V **V** extension.

We further expect that RISC-V will be used and extended in ways that we would never be able to anticipate. One of the most exciting prospects for RISC-V is its role as a research vehicle. We eagerly anticipate the free expression of other engineers' creativity in the form of novel ISA extensions and microarchitectural realizations.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. "Shared Memory Consistency Models: A Tutorial". In: *IEEE Computer* 29.12 (Dec. 1996), pp. 66–76. ISSN: 0018-9162. DOI: 10.1109/2.546611.

[2] Sarita V. Adve and Mark D. Hill. "Weak Ordering&Mdash;a New Definition". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. Seattle, Washington, USA: ACM, 1990, pp. 2–14. ISBN: 0-89791-366-3. DOI: 10.1145/325164.325100.

[3] *Alpha Architecture Handbook*. Digital Equipment Corporation. Maynard, Massachusetts, 1996.

[4] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks. "Architecture of the IBM System/360". In: *IBM J. Res. Dev.* 8.2 (Apr. 1964), pp. 87–101. ISSN: 0018-8646. DOI: 10.1147/rd.82.0087.

[5] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. "A 1.3-GHz fifth-generation SPARC64 microprocessor". In: *Solid-State Circuits, IEEE Journal of* 38.11 (2003), pp. 1896–1905. ISSN: 0018-9200. DOI: 10.1109/JSSC.2003.818146.

[6] *ANSI/IEEE Std 754-1985, IEEE standard for floating-point arithmetic*. 1985.

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

[8] ARM Ltd. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. 2014. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/.

[9] ARM Ltd. *ARM Architecture Reference Manual: ARMv8*. 2015. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0487a.h/.

[10] ARM Ltd. *ARM Holdings Strategic Report: Shaping the Connected World*. 2014. URL: http://ir.arm.com/.

[11] ARM Ltd. *ARMv6-M Architecture Reference Manual*. 2008. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419c/.

[12] ARM Ltd. *ARMv7-M Architecture Reference Manual*. 2014. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.b/.

[13] ARM Ltd. *Cortex-A15 Processor*. 2010. URL: http://www.arm.com/products/processors/cortex-a/cortex-a15.php.

[14] ARM Ltd. *Cortex-A57 Processor*. 2014. URL: http://www.arm.com/products/processors/cortex-a/cortex-a57-processor.php.

[15] ARM Ltd. *Cortex-M4 Processor*. 2009. URL: http://www.arm.com/products/processors/cortex-m/cortex-m4-processor.php.

[16] Arvind and Jan-Willem Maessen. "Memory Model = Instruction Reordering + Store Atomicity". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 29–40. ISBN: 0-7695-2608-X. DOI: 10.1109/ISCA.2006.26.

[17] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[18] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, 2014.

[19] Thomas W. Barr, Alan L. Cox, and Scott Rixner. "Translation Caching: Skip, Don'T Walk (the Page Table)". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France: ACM, 2010, pp. 48–59. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1815970.

[20] Gordon Bell and William D. Strecker. "Computer Structures: What Have We Learned from the PDP-11?" In: *Proceedings of the 3rd Annual Symposium on Computer Architecture*. ISCA '76. New York, NY, USA: ACM, 1976, pp. 1–14. DOI: 10.1145/800110.803541. URL: http://doi.acm.org/10.1145/800110.803541.

[21] Erich Bloch. "The Engineering Design of the Stretch Computer". In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts: ACM, 1959, pp. 48–58.

[22] John D. Bruner, Gary W. Hagensen, Eric H. Jensen, Jay C. Pattin, and Jeffrey M. Broughton. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCRL-97646. Lawrence Livermore National Laboratory, Nov. 1987.

[23] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. "Bringing Virtualization to the x86 Architecture with the Original VMware Workstation". In: *ACM Transactions on Computer Systems* 30.4 (Nov. 2012), 12:1–12:51. ISSN: 0734-2071. DOI: 10.1145/2382553.2382554.

[24] Brian Case. *Intel Reveals Pentium Implementation Details*. Microprocessor Report. Mar. 1993.

[25] Albert Chang, John Cocke, Mark F. Mergen, and George Radin. *Virtual memory address translation mechanism with controlled data persistence*. US Patent 4,638,426.

[26] Cobham Gaisler AB. *LEON3 Processor*. URL: `http://www.gaisler.com/index.php/products/processors/leon3`.

[27] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs". In: *Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. ISORC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 185–192. ISBN: 978-0-7695-4037-5. DOI: `10.1109/ISORC.2010.10`.

[28] Edsger W. Dijkstra. "Over de Sequentialiteit van Procesbeschrijvingen". In: (1962 or 1963).

[29] Embedded Microprocessor Benchmark Consortium. *Exploring CoreMark: A Benchmark Maximizing Simplicity and Efficacy*. 2009. URL: `http://www.eembc.org/techlit/coremark-whitepaper.pdf`.

[30] 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*. 1990, pp. 15–26.

[31] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. "Is SC + ILP = RC?" In: *Proceedings of the 26th Annual International Symposium on Computer Architecture*. ISCA '99. Atlanta, Georgia, USA: IEEE Computer Society, 1999, pp. 162–171. ISBN: 0-7695-0170-2. DOI: `10.1145/300979.300993`.

[32] Robert P. Goldberg. "Survey of virtual machine research". In: *Computer* 7.6 (1974), pp. 34–45.

[33] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. "The NYU Ultracomputer&Mdash;Designing a MIMD, Shared-memory Parallel Machine (Extended Abstract)". In: *Proceedings of the 9th Annual Symposium on Computer Architecture*. ISCA '82. Austin, Texas, USA: IEEE Computer Society Press, 1982, pp. 27–42.

[34] Torbjörn Granlund and Peter L. Montgomery. "Division by Invariant Integers Using Multiplication". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA: ACM, 1994, pp. 61–72. ISBN: 0-89791-662-X. DOI: `10.1145/178243.178249`.

[35] Linley Gwennap. *Cyrix M1 Design Tapes Out*. Microprocessor Report. Dec. 1994.

[36] Linley Gwennap. *Intel's P6 Uses Decoupled Superscalar Design*. Microprocessor Report. Feb. 1995.

[37] C.C. Hansen and T.J. Riordan. *RISC computer with unaligned reference handling and method for the same*. US Patent 4,814,976.

[38] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. "MIPS: A Microprocessor Architecture". In: *Proceedings of the 15th Annual Workshop on Microprogramming*. MICRO 15. Palo Alto, California, USA: IEEE Press, 1982, pp. 17–22.

[39] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-596-7.

[40] Maurice Herlihy. "Wait-free Synchronization". In: *ACM Transactions on Programming Languages and Systems* (Jan. 1991), pp. 124–149.

[41] Maurice Herlihy and J. Eliot B. Moss. "Transactional Memory: Architectural Support for Lock-free Data Structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA: ACM, 1993, pp. 289–300. ISBN: 0-8186-3810-9. DOI: 10.1145/165123.165164. URL: http://doi.acm.org/10.1145/165123.165164.

[42] Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, B. K. Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, Corinna Lee, Daebum Lee, Joan Pendleton, Scott Ritchie, David A. Wood, Ben Zorn, Paul Hilfinger, Dave Hodges, Randy Katz, John Ousterhout, and Dave Patterson. "Design Decisions in SPUR". In: *IEEE Computer* 19 (11 1986).

[43] "IEEE Standard for Ethernet - Section 1". In: *IEEE Std 802.3-2012 (Revision to IEEE Std 802.3-2008)* (2012), pp. 1–0. DOI: 10.1109/IEEESTD.2012.6419735.

[44] Institute of Electrical and Electronics Engineers. *IEEE Standard for a 32-bit microprocessor*. IEEE Std. 1754-1994. 1994.

[45] Intel Corporation. *i860 Microprocessor Family Programmer's Reference Manual*. Santa Clara, CA, USA, 1992. ISBN: 1-55512-135-7.

[46] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2015.

[47] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference*. 2015.

[48] International Organization for Standardization. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. 2011.

[49] International Organization for Standardization. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. 2011.

[50] Bruce L. Jacob and Trevor N. Mudge. "A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations". In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: ACM, 1998, pp. 295–306. ISBN: 1-58113-107-0. DOI: 10.1145/291069.291065.

[51] Stephen C. Johnson. "The I486 CPU: Executing Instructions in One Clock Cycle". In: *IEEE Micro* 10.1 (Jan. 1990), pp. 27–36. ISSN: 0272-1732. DOI: 10.1109/40.46766.

[52] W. Kahan. *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic*. 1997. URL: https://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

[53] Gerry Kane. *MIPS R2000 RISC Architecture*. 1st ed. Longman Higher Education, 1988.

[54] Gerry Kane. *PA-RISC 2.0 Architecture*. ISBN 978-0131827349. Prentice Hall, 1995.

[55] Michael Kanellos. *End of the line for HP's Alpha*. CNET. Aug. 18, 2004. URL: http://www.cnet.com/news/end-of-the-line-for-hps-alpha/.

[56] 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*. 1983.

[57] R. E. Kessler. "The Alpha 21264 Microprocessor". In: *IEEE Micro* 19.2 (Mar. 1999), pp. 24–36.

[58] Harvard University. Computation Laboratory. *A Manual of Operation for the Automatic Sequence Controlled Calculator*. Annals of the Computation Laboratory of Harvard University. MIT Press, 1946. ISBN: 9780262010849.

[59] Leslie Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computing* 28.9 (Sept. 1979), pp. 690–691. ISSN: 0018-9340. DOI: 10.1109/TC.1979.1675439.

[60] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators". In: *Proceedings of the 38th Annual International Symposium on Computer Achitecture*. ISCA '11. San Jose, CA, USA: ACM, 2011. ISBN: 978-1-4503-0053-7.

[61] Charles Lefurgy, Peter Bird, I. Chen, and Trevor Mudge. "Improving Code Density Using Compression Techniques". In: *Proceedings of the 30th Annual International Symposium on Microarchitecture*. 1997, pp. 194–203.

[62] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. "Value Locality and Load Value Prediction". In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: ACM, 1996, pp. 138–147. ISBN: 0-89791-767-7. DOI: 10.1145/237090.237173.

[63] Imagination Technologies Ltd. *MIPS Architecture for Programmers, Volume II-A: The MIPS64 Instruction Set Reference Manual*. 2015. URL: http://imgtec.com/mips/architectures/mips64/.

[64] Stephen Mccamant. *Efficient, Verifiable Binary Sandboxing for a CISC Architecture*. Tech. rep. MIT Computer Science and Artificial Intelligence Laboratory, 2005.

[65] MIPS Technologies, Inc. *microMIPS Instruction Set Architecture*. 2009. URL: http://cdn.imgtec.com/mips-documentation/login-required/micromips_instruction_set_architecture.pdf.

[66] MIPS Technologies, Inc. *MIPS R4000 Microprocessor User's Manual*. Mountain View, CA, USA, 1994.

[67] MIPS Technologies Inc. *MIPS64 Architecture for Programmers Volume I: Introduction to the MIPS64 Architecture*. 2010. URL: http://www.mips.com/products/architectures/mips64/.

[68] MIPS Technologies Inc. *MIPS64 Architecture for Programmers Volume IV-a: The MIPS16e Application-Specific Extension to the MIPS64 Architecture*. 2010. URL: http://www.mips.com/products/architectures/mips64/.

[69] James Montanaro, Richard Witek, Krishna Anne, Andrew J. Black, Elizabeth M. Cooper, Daniel W. Dobberpuhl, Paul M. Donahue, Jim Eno, Gregory W. Hoeppner, David Kruckemyer, Thomas H. Lee, Peter C. M. Lin, Liam Madden, Daniel Murray, Mark H. Pearce, Sribalan Santhanam, Kathryn J. Snyder, Ray Stephany, and Stephen Thierauf. "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor". In: *Digital Technical Journal* 9 (1 1997), pp. 49–62. ISSN: 0898-901X. URL: http://portal.acm.org/citation.cfm?id=268940.268945.

[70] S.J. Nadas and R.J. Pedersen. *Millicode register management and pipeline reset*. US Patent 5,226,164.

[71] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. "Practical, Transparent Operating System Support for Superpages". In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 89–104. ISSN: 0163-5980. DOI: 10.1145/844128.844138. URL: http://doi.acm.org/10.1145/844128.844138.

[72] OpenCores. *OpenRISC 1000 Architecture Manual, Architecture Version 1.1*. 2014.

[73] Oracle Corporation. *OpenSPARC Microprocessor*. URL: http://www.oracle.com/technetwork/systems/opensparc/.

[74] Oracle Corporation. *Oracle SPARC Architecture 2011*. Draft D0.9.6. May 2014.

[75] D. L. Osisek, K. M. Jackson, and P. H. Gum. "ESA/390 Interpretive-execution Architecture, Foundation for VM/ESA". In: *IBM Syst. J.* 30.1 (Feb. 1991), pp. 34–51. ISSN: 0018-8670. DOI: 10.1147/sj.301.0034.

[76] Andris Padegs, Brian B. Moore, Ronald M. Smith, and Werner Buchholz. "The IBM System/370 Vector Architecture: Design Considerations". In: *IEEE Trans. Comput.* 37.5 (May 1988), pp. 509–520. ISSN: 0018-9340. DOI: 10.1109/12.4602.

[77] *PALcode for Alpha microprocessors: System Design Guide*. Digital Equipment Corporation. Maynard, Massachusetts, 1996.

[78] David A. Patterson and Carlo H. Séquin. "RISC I: A Reduced Instruction Set VLSI Computer". In: *ISCA*. Minneapolis, Minnesota, USA, 1981, pp. 443–458.

[79] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. "Formal Virtualization Requirements for the ARM Architecture". In: *Journal of Systems Architecture* (Mar. 2013), pp. 144–154.

[80] Ken Popovich. *Alpha proved costly for Compaq*. ZDNet. July 3, 2001. URL: http://www.zdnet.com/article/alpha-proved-costly-for-compaq/.

[81] George Radin. "The 801 Minicomputer". In: *SIGARCH Computer Architecture News* 10.2 (Mar. 1982), pp. 39–47. ISSN: 0163-5964. DOI: 10.1145/964750.801824.

[82] Richard M Russell. "The CRAY-1 Computer System". In: *Communications of the ACM* 21.1 (1978), pp. 63–72.

[83] Alan Samples, Mike Klein, and Pete Foley. *SOAR Architecture*. Tech. rep. UCB/CSD-85-226. EECS Department, University of California, Berkeley, 1985. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1985/5940.html.

[84] Silicon Graphics International Corp. *SGI UV 3000 Data Sheet*. 2015. URL: http://www.sgi.com/pdfs/4555.pdf.

[85] Michael Slater. *AMD's K5 Designed to Outrun Pentium*. Microprocessor Report. Oct. 1994.

[86] Peter Song. "UltraSparc-3 Aims at MP Servers". In: (Oct. 1997).

[87] SPARC International, Inc. *The SPARC Architecture Manual, Version 8*. 1992. URL: http://sparc.org/technical-documents/#V8.

[88] SPARC International, Inc. *The SPARC Architecture Manual, Version 9*. 1994. URL: http://sparc.org/technical-documents/#V9.

[89] William Strecker. "VAX-11/780—A Virtual Address Extension to the DEC PDP-11 Family". In: *AFIPS Spring Conference*. 1978.

[90] Sven Stucki. Personal communication. Oct. 15, 2015.

[91] System Performance Evaluation Cooperative. *SPEC CPU2006 Benchmarks*. 2006. URL: http://www.spec.org/cpu2006/.

[92] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: ACM, 2010, pp. 463–468. ISBN: 978-1-4503-0002-5. DOI: 10.1145/1837274.1837390.

[93] Greg Tang. In: *Harvard Journal of Law and Technology* (Jan. 4, 2011). URL: http://jolt.law.harvard.edu/digest/patent/intel-and-the-x86-architecture-a-legal-perspective.

[94] *The Linux kernel source, version 3.14.29*. 2015. URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/log/?id=refs/tags/v3.14.29.

[95] James E. Thornton. "Parallel Operation in the Control Data 6600". In: *Proceedings of the Spring Joint Computer Conference*. 1964.

[96] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. *CACTI 5.1*. Tech. rep. HP Laboratories, Apr. 2, 2008.

[97] Linus Torvalds. Linux Kernel Mailing List. Apr. 13, 2009. URL: https://lkml.org/lkml/2009/4/13/122.

[98] Will Wade. *Lexra to move for dismissal of MIPS patent suit*. EE Times. Sept. 20, 2001. URL: http://www.eetimes.com/document.asp?doc_id=1143869.

[99] Andrew Waterman. "Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed". MA thesis. University of California, Berkeley, 2011.

[100] Andrew Waterman. *Spike, a RISC-V ISA Simulator*. 2015. URL: https://github.com/riscv/riscv-isa-sim.

[101] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Compressed Instruction Set Manual, Version 1.9*. Tech. rep. UCB/EECS-2015-209. EECS Department, University of California, Berkeley, 2015. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.html.

[102] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[103] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, 2014. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[104] Reinhold P. Weicker. "Dhrystone: A Synthetic Systems Programming Benchmark". In: *Communications of the ACM* 27.10 (Oct. 1984).

[105] F.C. Williams, T. Kilburn, and G.C. Tootill. "Universal high-speed digital computers: a small-scale experimental machine". In: *Journal of the Institution of Electrical Engineers* 1951.3 (1951).

[106] Kenneth C. Yeager. "The MIPS R10000 Superscalar Microprocessor". In: *IEEE Micro* 16.2 (Apr. 1996), pp. 28–40. ISSN: 0272-1732. DOI: 10.1109/40.491460.

[107] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–93. ISBN: 978-0-7695-3633-0. DOI: 10.1109/SP.2009.25.

Appendix A

User-Level ISA Encoding

This appendix contains the encodings of the instructions in the RV32G and RV64G user-level ISAs, described in Chapters 3 and 4.

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

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

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
000000000011		00000	010	rd	1110011	FRCSR
000000000010		00000	010	rd	1110011	FRRM
000000000001		00000	010	rd	1110011	FRFLAGS
000000000011		rs1	001	rd	1110011	FSCSR
000000000010		rs1	001	rd	1110011	FSRM
000000000001		rs1	001	rd	1110011	FSFLAGS
000000000010		00000	101	rd	1110011	FSRMI
000000000001		00000	101	rd	1110011	FSFLAGSI

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