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

PREVIEW

Committee in charge:

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

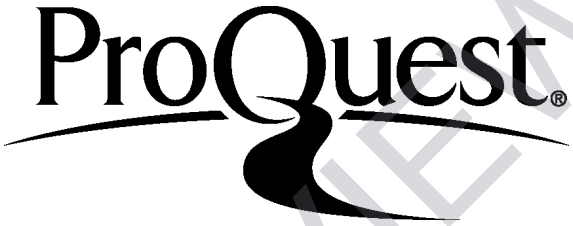
Spring 2016

ProQuest Number: 10150769

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest 10150769

Published by ProQuest LLC (2016). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

PREVIEW

Design of the RISC-V Instruction Set Architecture

Copyright 2016
by
Andrew Shell Waterman

PREVIEW

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.

PREVIEW

Contents

PREVIEW

PREVIEW

List of Figures

PREVIEW

PREVIEW

List of Tables

PREVIEW

PREVIEW

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.

PREVIEW

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.

PREVIEW

Chapter 1

Introduction

PREVIEW

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.

PREVIEW

Chapter 2

Why Develop a New Instruction Set?

PREVIEW

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.

PREVIEW

- 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

PREVIEW

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

PREVIEW

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.

PREVIEW

- 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.

PREVIEW

- 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.

PREVIEW

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.

PREVIEW

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.

PREVIEW

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.

PREVIEW