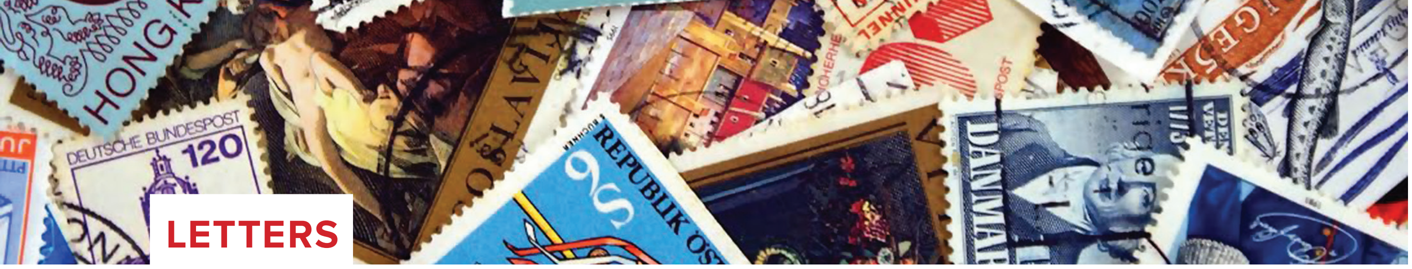

LETTERS

ANOTHER TAKE ON RISC

To the editor:
After reading the article "Reduced Instruction Set Computers Then and Now," by David Patterson (December 2017, pp. 10–12), I wanted to share another perspective. In the 1980s, nobody knew what was going to happen in the computer industry over the ensuing 20 or 30 years. It went almost without saying that RISC ISAs (instruction set architectures) would take over the world, due to their predicted major advantages over the ugly old CISC chips such as the x86. Even within Intel, there were some powerful voices echoing that theme. We can now look back and see that that's not what happened.

Patterson suggests that there have been no new general-purpose CISC ISAs since RISCs first appeared. Even if true, I'm not sure what that proves. But I am sure that the archetypical CISC, the x86, was well established before RISC came along. In fact, it not only withstood but thrived in competition with an array of RISC attackers: PowerPC, Alpha, MIPS, Sparc, PA-RISC, M88K, AM29K. The x86 has been the dominant computer architecture in everything from servers to laptops for more than 30 years. Nobody would argue that this success was a result of some secret advantage of the x86 ISA per se. The very least one should conclude from this is that ISAs are evidently not all that matters.

Patterson asserts that "the hardest part of computer design is control." In a way, that's arguably true, but he essentially goes on to equate control complexity to microcode, in which case the assertion is inarguably false. As part of these architecture efforts, I personally supervised the microcode teams for the Pentium chips of the 1990s; out of teams with 500–900 designers, fewer than 10 of them were in the microcode group. Although microcode isn't trivial, design engineers don't consider it to be harder than many other aspects of modern industrial microprocessor design. If you want to win on the world stage, it's *all* hard no matter which ISA you're implementing.

I agree with Patterson that x86s surpassed RISC performance in the 1990s: specifically, with the P6/Pentium Pro in 1995. But I certainly do not agree that somehow the P6 achieved its supremacy by using "RISC ideas." Nonsense. P6 was based on speculation, out-of-order execution, deep (disjoint) pipelines, good branch prediction, superscalar microarchitecture, register renaming, fast clocks, and the concept of micro-ops. Every one of those ideas was already in use in machines prior to RISCs except for micro-ops, and that one we invented ourselves. Bottom line, we dragged an ugly ISA (x86) up a steep hill and beat the best of the RISC competition. If ISAs are so important, how was that possible?

Next, Patterson switches to unit sales volumes; he points out that the volumes of ARM cores being shipped today dwarf the x86 volumes being sold by Intel and AMD. He implies that this was somehow an inevitability, that there's a "renewed need in the post-PC era for simpler ISAs." But that's a leap to a conclusion for which he has provided no evidence. I know from first-hand experience that from 1996 through at least 2003, Intel simply refused to try to compete at the low-power end of the CPU space. As their chief architect for most of that period, I tried hard to get them interested in designing a very-low-power x86 for those emerging markets. Upper management refused, mainly on the grounds that the profit margins in those markets were an order of magnitude less than they already enjoyed. In effect, they refused to compete. So we cannot know whether x86 would have been able to compete squarely in these high-volume markets; by the time new management tried, it was too little, too late.

I have a RISC-V, it looks like a good architecture to me, and per Patterson's description, it has benefited from 30+ years of collective experience. For me, the upshot of the RISC/CISC debates of the 1980s wasn't to try to establish which was "best" (whatever that might mean). For me, the RISC research represented a break from the quasi-religious approaches that preceded it, such as "narrowing the semantic gap" between ISAs and assembly programmers, or quixotic quests for architectural purity such as Intel's 432. The RISC researchers quantified the performance of competing ideas, and by example forced the entire field onto a numerical basis for making progress. And that lesson, we *did* take to heart in designing Intel's chips.

Bob Colwell
bob.colwell@gmail.com

~~The~~ Author's Response:
I felt a wave of nostalgia when I read Bob Colwell's letter questioning my assessment of RISC in my short retrospective. His letter reminded me of stories of veterans of the US Civil War in the same retirement home arguing how they could have won critical battles with a few changes in tactics or luck.

Colwell and I are veterans of the RISC vs. CISC conflict of the 1980s. He signed up while a grad student to the CISC side,[1,2] and then did his best in his dissertation to find virtues in Intel's infamous 432 processor, certainly the most complicated microprocessor of the 1980s.[3] And he's the father of the P6, the most successful x86 microarchitecture.

Colwell argues that it's "nonsense" that the P6 is based on RISC ideas. RISC machines were the first microprocessors to:

- Have split instruction and data caches;

- Use the efficient five-stage pipeline, enabled by the split caches;
- Have a superscalar microarchitecture, which allowed fetching and executing multiple instructions per clock cycle;
- Use "superpipelining," a much deeper pipeline than five stages, enabling higher clock rates; and
- Offer multilevel caches on chip, which help overcome the performance bottleneck of long latency to memory especially as processor clock rates increased.

Although some of these ideas first appeared in larger computers, RISC engineers were the first to make them work within the constraints of a single chip. Using the industry standard SPECint benchmark as the measure, these innovations made RISC microprocessors the fastest in the 1980s and 1990s.[4]

Intel and AMD designers rose to the challenge. The first step was to translate the variable length and variable time x86 instructions into simpler fixed-length instructions (similar to what DEC VAX engineers did in the 1980s). Intel calls them *micro-operations*, and AMD calls them *RISC operations*. They then used all the ideas above. The RISC and x86 designers added out-of-order execution at about the same time, which comes from a high-end IBM S/360 mainframe of the 1960s. Intel and AMD had larger teams and access to better semiconductor processing than the RISC companies, and, with excellent engineering, x86 microprocessors became the fastest starting in the 2000s. As I said in my retrospective, by the end of the PC era, x86 had clearly won.

Like the Civil War veterans reliving battles, Colwell thinks Intel could have won the PostPC era as well with a small change in strategy. In his view, the flaw isn't the inherent complexity overhead of the x86, it was simply waiting too long to try. UC Berkeley's parallel computing research was cosponsored by Intel from 2007 to 2013, and we learned that the top priority of the Intel CEO Paul Otellini during those years was getting x86 into the mobile market. Intel finally gave up on that goal in 2017. Many blame the overhead in design, area, power, and verification of the complex x86 instruction set.

ARM ("Advanced RISC Machine") has virtually the same monopoly on mobile devices now that x86 had on PCs; it recently celebrated shipment of its 100 billionth chip. Surely fewer than 5 billion chips use the older x86 instruction set.

Indeed, the biggest competitor in the future is likely not x86, but the free and open RISC-V (www.riscv.org). It's a simple, elegant instruction set architecture. (Even Colwell likes it.) It competes as an open alternative to the proprietary instruction sets, much like open Linux competes with Microsoft Windows. Time will tell if RISC-V will have as much success in processors as Linux has had in operating systems, but that is the long-term goal.

Sadly, Otellini recently passed away. One obituary noted as his biggest regret that he sold off an ARM processor design and team that Intel had acquired from another company.[5] Had they kept the product, Steve Jobs might have used it in the first iPhone in 2007, since it was clearly the best ARM processor of that era. (In fact, Apple eventually hired the engineers who designed those processors.) Apple still uses ARM in iPhones and iPads. Imagine how much rosier Intel's future would be if it was the main supplier of chips for both the PC era *and* the Post-PC era!

I'd like to thank Colwell again for the trip down memory lane that let me recall the RISC vs. CISC battle of our youth. We both agree that the winners have varied over time. I believe the field benefits from the frank and open exchanges like the ones we've participated in now and in the past.

Architecture debates have traditionally been settled ultimately by companies spending billions of dollars developing products on both sides of an issue, and letting the marketplace determine the winner. Of the 20 billion 32-bit and 64-bit processors to be shipped this year, 99 percent will be RISC. I'll leave it to the readers to decide whether RISC or CISC dominates today, and whether the reason is inherently technical or simply better business decisions.

David Patterson
pattrsn@cs.berkeley.edu

REFERENCES

1. R.P. Colwell et al., "Peering through the RISC/CISC Fog: An Outline of Research." *ACM SIGARCH Computer Architecture News*, vol. 11, no. 1, pp. 44–50.
2. R.P. Colwell et al., "Instruction Sets and Beyond: Computers, Complexity, and Controversy," *Computer*, vol. 18, no. 9, 1985, pp. 9–18.
3. R.P. Colwell, "The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems," PhD dissertation, Carnegie Mellon University, 1985.
4. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th Edition, 2018, Elsevier. [see Figure 1.1]
5. D. Clark, "Paul S. Otellini, Who Led Intel and Saw It Grow Even More, Dies at 66," *The New York Times*, 3 October 2017.

IN DEFENSE OF PROGRAMMING LANGUAGE RESEARCH

To the Editor:

I read the opinion piece "On Methodological Irregularities in Programming Language Research," by Andreas Stefik and Stefan Hanenberg in the August 2017 issue (pp. 60–63). The authors question programming language (PL) design conferences such as PLDI, OOPSLA, ICFP, and ECOOP for a lack of "rigorous evidence standards like those in other sciences"—standards such as randomized controlled trials for evaluating the effectiveness of an intervention. The authors conclude by calling for the imposition of strict reporting standards in software engineering conferences and journals, like the Consolidated Standards of Reporting Trials (CONSORT) statement (www.consort-statement.org) in medicine and the What Works Clearinghouse (WWC) guidelines (http://ies.ed.gov/ncee/wwc/Docs/referenceresources/wwc_procedures_v3_0_standards_handbook.pdf) in education.

Stefik and Hanenberg's recommendations are reasonable for a certain class of software engineering research—specifically, for research aiming to show that a given intervention, such as the adoption of a particular language or technique, has certain economic or pedagogical benefits. Randomized controlled trials are the gold standard of evidence for this kind of claim, and repeatability requires careful attention to sample size and selection, control of confounding factors, and so on. Robust empirical claims require robust empirical evidence.

However, it is somewhat narrow-minded to expect all PL research to follow this empirical pattern. Not all scientific claims concern the effectiveness of an intervention on human subjects, and so not all papers are suitable targets for standards such as CONSORT and WWC. In particular, it is not appropriate to criticize conferences such as ICFP for a "lack [of] empirical foundation" when most of the papers published there do not make empirical claims. One might as well criticize Einstein's "Does the Inertia of a Body Depend upon Its Energy Content?," Turing's "On Computable Numbers, with an Application to the Entscheidungsproblem," and Watson and Crick's "A Structure for Deoxyribose Nucleic Acid," for such "methodological irregularities."

There is much more to PL research than empirical claims of the kind that Stefik and Hanenberg have in mind—for example, mathematical semantics of language features, verification and proof of correctness, type systems to allow the formal statement of certain properties of programs, static and dynamic analysis to check those properties, systems building and engineering design, compiler and performance optimizations, implementation techniques, and so on. For a longer discussion, see for example the "PL Enthusiast" blog post by Michael Hicks (www.pl-enthusiast.net/2015/05/27/what-is-pl-research-and-how-is-it-useful).

Jeremy Gibbons,
Jeremy.Gibbons@cs.ox.ac.uk

Authors' response:
Jeremy Gibbons emphasizes that programming language researchers work on many aspects of language design. Sometimes, these considerations are fleshed out mathematically, like imagining a new semantics or working with verification, type systems, or something else. By declaring that such topics do not require empirical validation, Gibbons implies that programming language constructs are pure mathematical constructs, ignoring that the vast majority of programs are written by people.

Other scientific fields balance mathematical formalism and empirical data. In physics, including with Einstein's work, scholars have gone to great lengths to validate or refute the theories. The empirical work regarding Einstein's theories is quite famous and important. Other exemplars involving Faraday, Maxwell, and others also weave a more accurate story about how formalism and empirical data have been balanced over time in a field like physics. Most fields have such a balance for a variety of reasons. For example, nature is not guaranteed to agree with a formula or explanation imagined by a scholar.

Second, the focus of programming language research is generally no longer on the development of a calculus to understand what computation is, as it was during Turing's time. There is more to learn, but we now know quite a bit about computation. Programming language research today, amongst other things, is often about providing tools and techniques to make programming, or programs, more efficient or less error-prone. Further, many kinds of programming languages have mathematically valid semantics, or sound type systems, but this does not mean their benefits and costs to society are known. Given these costs are in the billions and they impact millions of people, empirical evidence is needed to sort out the facts.

Finally, Gibbons argues that not all programming language researchers need to use empirical evidence or measure the human impact, which we agree with. However, empirical data is hardly an all-or-nothing proposition. We cited evidence that the amount of data gathered by the programming language research community on human factors approaches zero. While it is clear that "for all" is not necessary, pointing out meticulous observations from other scholars that suggest the value is "near zero" seems pretty reasonable. After all, any group or person that uses a programming language should be aware that little or no human testing has occurred in the language community. Acknowledging the observable fact that we lack evidence today is the first uncomfortable step in recognizing that change is needed for the benefit of the field tomorrow.

Formal statements are part of the picture, but they tell us little about the impacts specific technologies have on people or communities. Put another way, yet another proof of something like type soundness does not give us enough information about the impact of a language, feature, or technique. In our view, a larger and much more diverse collection of evidence-based ideas is needed.

Andreas Stefik
stefika@gmail.com
Stefan Hanenberg
stefan.hanenberg@uni-due.de