This Architecture Tastes Like Microarchitecture

Curtis Dunham
Jonathan C. Beard

Original Article:
Curtis Dunham, Jonathan C. Beard
This Architecture Tastes Like Microarchitecture

Published in:
The 2nd Workshop on Pioneering Processor Paradigms (WP3) 2018 (held in conjunction with HPCA 2018)

This Architecture Tastes Like Microarchitecture

Curtis Dunham and Jonathan Beard
Arm Research
Austin, Texas, USA
{*first.last*}@arm.com

***Abstract*—Instruction set architecture bridges the gap between actual implementations, or microarchitecture, and the software that runs on them. Traditionally, instruction sets were a direct reflection of the hardware resources and capabilities. The two drifted apart in the rise of CISC and its microcoded implementations. In the 1980s, the RISC movement reasserted the philosophy that the two should correspond, and that microcode was a less desirable approach. Nevertheless, time has shown that the natural tendency in industrial designs is to treat the instruction set as an abstraction.**

In this paper we review, with several decades of hindsight, an early RISC proposal in the form of the original MIPS architecture. While we find that the RISC movement left a legacy congruent with its philosophy, the specific techniques proposed in this seminal work were considerably more aggressive and did not succeed. In our investigation, we find that RISC's impact on microarchitecture should be contrasted with its impact on ISA design, where a promising and under explored approach is to specify and therefore assume less about how the machine works, not more. To that end, the authors review several competing ISA design proposals from others; some that are aligned with the idea that less detail about the machine is actually more and others, such as transport triggered architecture, that take machine detail to the extreme.

I. Introduction

Broadly speaking, this position paper is about the design of instruction set architectures (hereafter ISA(s)). In summary, the strongest tendency is to specify the ISA in a way that exposes, rather than abstracts over, microarchitectural details of the naturally envisagable implementations. While this bias seems natural, we argue that it was further encouraged, to our detriment, by the RISC philosophy originally espoused by academics throughout the 1980s. By contrast, the trend in industrial implementations (with some notable exceptions) is often to maintain the ISA as an abstraction and to conceal performance-enhancing indirection behind that abstraction. The tendency of over-specifying ISAs seems natural and benign in retrospect, but can it be counteracted? Our ultimate aim is to convince the reader that an ISA best suited for both market factors and technical trends, both foreseeable and otherwise, would maximize its flexibility and therefore applicability by specifying as little as possible about its implementation, and we should explore what benefits arise when these familiar constraints are lifted.

Our presentation is divided in two parts, covering what some might consider mis-steps in ISA design of the past, followed by a discussion of what are perhaps some brighter alternatives. This begins with a critical review of the original MIPS architecture in section II, as it represents a very early attempt to (re-)lower the abstraction level of the ISA. Section III continues in the same vein but focuses on specific lamentable decisions in other successful architectures, noting that they tend to focus on low-level aspects of the machine.

A. ISAs: An interface is forever

To a first approximation, code never dies, but machine implementations last but a few years. As a result, software retains remnants of machine guts from ages past without an expiration date. Consider Lisp's *car* and *cdr* accessors of the *cons* cell (pair) constructor, which get their names from assembler macros that extracted the address and decrement fields, respectively, of a memory word on the IBM 704. While modern machines have abandoned such memory organizations, this notation has sixty years of well-earned longevity at the time of this writing, with no foreseeable end.

While machine interfaces are made low-level in order to extract performance, the specific machines to which they refer are constantly being improved, often obsoleting the interface along with the implementation. While section III gives some obvious examples, section IV presents the hypothesis that certain often ignored design fixtures of modern ISAs belong to this same category. The authors suspect this is due to a misinterpretation of the RISC legacy, which championed low-level interfaces like load/store memory architecture.

B. Towards a desirable "forever interface"

Time and time again, attempts to achieve performance through machine-describing ISAs eventually lose to implementations that achieve comparable (or better) efficiency with less effort. The present authors pose an alternate philosophy of instruction set design: an ISA should serve the computations, not the machines. In wanting to describe computations of all kinds, some computations find themselves suffering from incompatibilities with the machine architecture. In the penultimate section V the historical emphasis gives way to an examination of a major problem facing the architecture community today, and how our perspective provides direction towards a solution. As we conclude in section VI, we hint at the long term vision for this work: ISAs that break down the division between the silos of CPU and GPU, between reuse-optimized architecture and throughput architecture.

II. MIPS: Microprocessor without Interlocking Pipeline Stages

We find the MIPS treatise by Hennessy, *et al.* [1] to be representative of the philosophical thrust of the early

RISC movement. Nota bene, while RISC stands for *Reduced Instruction Set Computing*, the movement was about more than just reducing the instruction set. Its advocates wanted to regain performance by being able to speak directly to the microarchitectural structures on the chip, namely the individual pipeline units (not just ALUs) and registers, which was the purpose of those reduced instructions. Such a machine would need no microcode nor would it need hardware interlocks, because compilers would automatically handle all of the machine's tricky aspects. While unsurprising in retrospect that this approach didn't pan out, we detail some of the reasons below.

A. Pipeline interlocks

The MIPS originally had no pipeline interlocks and depended on the software stack to properly avoid any incorrect behaviors. This approach is untenable for many reasons, but we will focus on microarchitectural complexity, security concerns, and intellectual property protection. By contrast, today any such code is part of the implementation-specific microcode so its use is restricted to fully validated circumstances. We consider the industrially adopted practice to be superior.

In order for MIPS machine code to perform microengine-level coordination, it must have perfect knowledge of the timing of the machine. In fact, [1] describes a scheme that could be likened to VLIW in control philosophy for how it statically schedules machine resources, except that the machine was so simple that the instructions were not very long. The increased use of aggressive speculation made this approach all but impossible, however. A deeply pipelined superscalar machine with cache memories and dynamic branch prediction can stall under many circumstances, so mainstream CPUs are dynamically scheduled by necessity. (While a VLIW-style microarchitecture is not impossible, the known implementations require very sophisticated translation layers, far more complex than typical microcode [2]. Eschewing hardware interlocks this way is certain to run afoul of the "no microcode" tenet described in the next section.)

The original MIPS architecture's ability to operate without pipeline interlocks depended directly upon the machine code being completely tuned for deep microarchitectural properties of the implementation. This circumstance brings with it concerns that certain code sequences would be unsafe, *i.e.,* code flaws could potentially lead to unrecoverable corruption or crashes. In today's security theater where an attacker can get a browser to emit and run specific machine code sequences (consider asm.js [3] or WebAssembly [4] plus RowHammer [5] or Spectre [6]), such an approach is a non-starter. The microarchitecture and associated circuitry simply cannot depend on software to maintain its own internal consistency; rather, the ISA must attempt to be an impenetrable barrier between the software and the underlying implementation regardless of inputs (while the authors acknowledge that this barrier is more likely to continue to be more of a "Maginot Line" [7] versus a perfect defense).

Finally, most vendors have strong incentives to protect their trade secrets. Exposing the deep details of a modern machine to software runs counter to this goal. While the authors do not advocate opaque designs such as Intel's Management Engine [8][9], neither can designs that are fully transparent to software control be justified. The trend towards heavy operating system-like micro-engines that run on cores is worrisome in that they open up the potential for malware at the μISA level. Consider for example the recent disclosure that the microcode format and updating procedure on recent AMD CPUs have been reverse-engineered [10], demonstrating the potential for fascinating and frightening μISA malware. The simple conclusion is that delegating pipeline interlocking and related concerns to software is a non-starter. The micro-engine's operation must be internally and securely managed.

B. No microcode

The MIPS design explicitly gave software direct control over the "micro-engine." To adhere to this philosophy most strongly, the binaries generated for one revision of the architecture could not necessarily be considered compatible with the next.

We are careful to not assert that this is undesirable *a priori*, but rather that it is a very impactful decision to make. During that era, commodity compilers of the quality we currently enjoy in the form of gcc [11] and LLVM [12] did not exist, nor did a large body of commercially viable software freely available in source code form. Demanding that programs be compiled anew for every new version of the machine was simply an untenable proposition, and even in today's software ecosystem it would face troublesome friction.

Consider by contrast long-lived ISAs whose vendors have consciously focused on backwards compatibility: IBM's 360/370 family and Intel's x86 and x86-64. Maintaining the interface carries market advantages (most notably the software ecosystem that inevitably must be built around any ISA for it to be successful), while new performance enhancing technologies hide behind the ISA interface and further its viability. This necessitates a separation between ISA and μISA and with it some amount of microcode. We notice that this points in exactly the opposite direction of the MIPS micro-engine philosophy. What was intended by MIPS to be the only layer of the micro-engine, is now simply the first layer of indirection after the assembly language. To borrow words from David J. Wheeler, "We can solve any problem by introducing another level of indirection." Micro-code is that second layer of indirection, solving the problem of exposing a static layer (ISA) that can't easily be changed. The introduction of microcode is inevitable in any long lived architecture, no matter how true to the original MIPS philosophy its designers intend to be. Again, we place our vote with the accepted industry practice: use of microcode is inevitable and carries many advantages.

So not only is microcode inevitable for correctness and security reasons, it is also necessary for maintaining the ISA interface across machine revisions.

C. RISC: a retrospective

It is quite revealing to carefully examine what was being proposed under the RISC banner in 1981. It is common today to hear sentiment such as "in microarchitecture RISC won, but in ISAs CISC won," supported by the observation that today's designs tend to have μISA back-ends processing simple μ-instructions that seem to embody the RISC ideal. To engage in such thinking unfairly allows a disconnection between RISC's hardware and software philosophies, as there was no such separation. RISC's goal of directly exposing the hardware in the name of performance is a pattern that repeats quite often [13][14][15]. Furthermore, the modern perspective discounts the extent to which the original MIPS design concerns were not solved in the way originally advocated; *i.e.* microarchitectures today bear little resemblance with this MIPS work. RISC adherents championed simple ISAs with no microcode, but in an ironic twist of fate, that design ethos is now embodied in the microcode layer. This is why they are given credit for microarchitecture, but as we have argued, the microarchitecture proposed in this early work was not viable. Considering the early days of RISC, perhaps a more accurate statement would be that "RISC won the μISA". What is considered RISC microarchitecture was explored concurrently by designers of both RISC and non-RISC ISAs.

III. Reflections on modern mis-steps

Regrettable mistakes in ISA come in many flavors, but one common aspect is exposing the machine's internal operations to the interface. We first discuss two ways that ISAs have attempted to make concessions for branch instructions. Next we describe an implementation detail of an early Arm processor that became a long-lived standard for backward compatibility. Finally we reflect on Intel's series of SIMD extensions.

A. Delay slots and branch hints

Branch instructions pose challenges of nightmarish difficulty for computer architects. Branch delay slots and branch hints are two ISA-exposed techniques intended to mitigate such difficulties. Neither of these techniques are considered appropriate any longer, as engineers found better solutions—transparent ones, to be clear.

A branch delay slot refers to one or more instructions after a branch which are still executed regardless of whether the branch is taken. This gives the processor some extra work to do when it would otherwise be waiting for the pipeline to refill. The MIPS and SPARC architectures, among a few others, employ the technique. This practice has been abandoned in modern architectures as it is better handled by increased aggressiveness and improved speculation in the processor front-end.

Branch hints are annotations in the instruction stream that indicate a branch's expected bias. They could be encoded in normal branch instructions or in no-ops or prefixes. Their intent is to reduce front-end stalls due to branch mispredictions. The useful life of this concept was similarly short: since conditional branch predictors perform better than static predictions, these branch hints are ignored in nearly all cases. Furthermore, such hints may increase code size, as an example, it costs 1 byte on x86) and because the processor must decode the instruction to respond to the hint, a small stall is unavoidable for a cold taken branch. The best practice is to make the fall-through code path the common case. For loop back-edges, cold backwards branches are often predicted taken in modern designs anyway. This way I$ resources are conserved and stalls due to cold front-end components such as the BTB and branch predictor are minimized. As further evidence, consider that GCC's response to branch hint intrinsics is to reorder code, but not emit branch hints.

B. Arm ISA's excepting instruction offsets

From at least version four of Arm's 32-bit ISA, now dubbed AArch32, when an exception is taken from user to supervisor privilege, the PC at the time of the fault is stored in the Link Register *with an offset* that depends on the type of exception. How could such a design come about? A natural hypothesis adopted by the present authors is that the initial implementation of the architecture in fact applied no offset at all; rather, the behavior was merely a timing artifact of how much further ahead the fetch stage's PC would be when the exception was detected later in the pipeline. One might say that exceptions were not precise in the original architecture, but could be made precise with software—a very RISC-like compromise! Over twenty years later, this imprecise exceptions behavior still remains for backwards compatibility. For exception behaviors introduced later, the Link Register value is completely precise.

This is yet another example of an architecture exposing a transient microarchitectural detail in the ISA. Unfortunately, like so many such well-intentioned compromises made early in the life of an ISA, it changed the trajectory by which future versions of the ISA had to match, whether or not the implementation congruence still existed, the probability of which tends to only ever decrease with time. Despite its minor severity, this quirk demonstrates the impact of choosing an expedient engineering solution visible to the machine's interface: in all likelihood, the interface-level decision is irreversible, but the next product iteration could take nearly any desirable approach behind the interface.

C. Intel SIMD extensions

While the x86 ISA has many examples of abandoned, vestigial features, its long history of SIMD extensions offers a lesson of a different sort. In particular, consider the many different ways one might encode integer addition of 32-bit words: starting from a single scalar `ADD` instruction, the `PADDW` instruction provided two-, four-, eight-, and sixteen-wide integer addition in the subsequent MMX, SSE2, AVX, and AVX-512 extensions, respectively.

Clearly the philosophy being demonstrated is similar to that of RISC, with the ISA directly referring to specific implementation details, such as the width of the vector unit and a register file sized to store vectors of that length.

The Cray-1 architecture [16], by contrast, represents a smoother evolutionary path, as it expresses vector computations of arbitrary length. A computation thus encoded needs no re-encoding when a later implementation provides more advanced capabilities, nor would said implementation need any extensions to its decoding circuitry. We offer this as an example of a weakness of the RISC approach, where directly exposing the capabilities of the latest machine revision seems very important in the short term, then often becomes regrettable baggage for all parties involved.

IV. The load-store legacy

A. Architecture and microarchitecture

Over the last half-century, there have been many attempts to redefine the relationship between architecture and microarchitecture. With the premise that the MIPS philosophy strikes some middle ground, then the extremes are formed by the likes of Transport Triggered Architecture (TTA) [13] and at the other end of the spectrum the Register-less Architectures (RLA) [17]. The TTA philosophy is that the programmer or compiler can decide exactly what instructions go to which functional units more efficiently than the microarchitecture can. As an example, taken to the extreme, this means that the instruction stream itself is devoid of arithmetic logic unit (ALU) operations and consists solely of data movement operations to locations that are the functional units. This directly exposes the microarchitecture itself, making binaries extremely non-portable. RLAs on the opposite end of the scale often start with a RISC-like base and remove the register moves. In theory, today's high speed caches are close to the speed of the register file. In a sense, the registers themselves are simply another smaller cache that is managed via compiler instructions which are load/store operators. The idea of the RLA is that instead of using load-to-register before computation, use the memory address itself directly (with appropriate concessions made for things like stack pointers, etc.). While demonstrating some performance advantages over register architectures, the potential for variable length instruction encoding and the resulting size of the binary (large), the concept of RLA has yet to become mainstream.

B. Memory-to-memory architecture

Patterson and Hennessy [18] state that an ISA should have a small number of operands that reference fast register memory. The first argument given for limiting the number of registers is the ability to clock these memories high enough to be effective. Secondly the authors reference the nice power of two encoding that 32 registers provide within the encoding space. This, however, hasn't stopped architectures from increasing the number of architected registers (as high as 128 for Itanium [19]). Many have advocated for so called "registerless" architectures which ostensibly encode memory addresses as operands for each instruction. The authors of this paper chose the phrase "so called" because most implementations of "registerless" architectures still employ fast registers, they simply hide them from the exposed programmer interface, choosing instead to manage the register space within the microarchitecture. If the register mapping problem [20] is viewed as simply solving a temporal encoding efficiency problem, then the memory-to-memory architectures can be viewed as having infinite registers (moving the encoding overhead to the microarchitecture) whereas the register-memory architectures can be viewed as having a very limited fast memory space extended by main memory (essentially a scratch-pad memory optimized by the compiler). The consequence of encoding a larger number of registers, as large as the addressable memory space (as in memory-to-memory architectures) is that the instruction encoding must be large as well to accommodate the memory addresses as operands. Conversely, these architectures can eliminate explicit load and store instructions which has the effect of reducing the overall number of needed instructions. Memory-to-memory architectures [21], such as PERL [17], embody this type of "registerless" system, however, to date memory-to-memory architectures have yet to catch on outside of embedded systems (see ATmega16 as an example [22]). A potential advantage of memory-to-memory architecture is that it enables architects to innovate around the register management while removing the long term limitations that fixing the number of registers within the ISA can have (*i.e.* once it is in the ISA, it is essentially permanent).

V. Raising the level of abstraction

A. On tasting like microarchitecture

What does it mean to, as we assert tongue-in-cheek in the title, "taste like" microarchitecture? As we have described up to this point, the RISC philosophy espouses an ISA where fundamental abstractions are shaped to match the outlines of the microarchitecture. Adherence to this motivation continues to the present day. Therefore the hardware-software interface has a flavor, a taste, of microarchitecture. Is this so bad? In a word, yes; we claim that this emphasis is simultaneously antiquated, at odds with solutions to real problems facing our field, and unnecessary. When considering the period of rapid evolution that microarchitecture is about to face with the end of lithography scaling, abstractions that are free from underlying microarchitectural influence are critical to minimizing future disruption.

B. Killer microseconds

The *killer microsecond problem* [23] is the observation that decreasing I/O wait times are fast approaching the latency of a context switch, thereby challenging the efficacy of time-sharing itself, a technique used effectively for over a half century [24]. For many decades, these delays were millisecond-scale while the context switch cost was microsecond-scale. How might the taste of an architecture be relevant to this problem?

It is helpful to examine the activities undertaken by the system on a context switch and how they are accomplished. In a modern time-sharing system like Linux, the flow of operations is as follows: the privilege level is changed to supervisor, the user thread's context (namely its architectural

state, *i.e.* the values stored in the register file(s)) is stored to memory, a scheduler decides what thread to run next, the context-swapping procedure is repeated to restore the context of the chosen thread, and the privilege level is returned to user level.

For clarity, we now point out the respective actors in the previous paragraph, which was intentionally written in the passive voice. Since our machines inherit the load-store legacy, software, specifically the operating system, performs the fundamental register-copying task of context switching. In fact, the ISA is *defined* such that the machine explicitly exists as a separate entity that programs must be multiplexed in and out of, rather than the machine model itself being defined as virtual, or even self-virtualizing and self-multiplexing. Therefore we have left software with the responsibility of encapsulating independent computations into processes or virtual machines. In the latter case, some architectural mitigations have been implemented, but such solutions still leave much to be desired.

C. An encouraging direction: stateless ISAs

As we alluded to in section IV, the load-store legacy and its register-centric machine model have alternatives worthy of further examination. We will briefly describe a collection of ideas in the direction of "registerless" architectures. To be clear, we are interested primarily in the interface and how it implicitly dictates implementations; we are not also proclaiming the obsolescence of register-based microarchitecture, as most techniques there are more generally applicable.

Suppose we took the approach of mapping the register file to memory, following Oehmke's work on *virtual context architecture* [25][26]. What we now find is that we have only gone halfway: while the register state has an architecturally defined backing store with a natural, hardware accelerate-able scheme for context switching, we also unnecessarily separated the natural working set of the program into two places in memory, namely the register backing store and the stack. It is not clear that this separation provides any real value when considering the envisioned techniques for a stack-centric "registerless" architecture (note that we do not consider this a stack architecture, but rather a memory-memory architecture with practical operand encoding).

One of the primary benefits of register operand encodings is the clear dataflow relationship between producers and consumers; only memory operations are subject to complex disambiguation schemes, while instructions with only register operands can proceed directly through register renaming to get their physical register/dataflow tag. We will explore some reasons why this advantage is likely overrated.

For maintaining the performance expectations established by modern CPU architectures, we will focus on the ability to connect producers with consumers back-to-back, or in consecutive clock cycles. This is not a feature provided by registers, whether architectural or microarchitectural, but rather an effective dataflow tagging scheme, a bypass network, and one or more schedulers. We expect that the same is no less achievable with only memory operands, albeit with the burden of encoding and mapping dataflow via tags/registers now squarely in the realm of microarchitecture.

Starting in the 1990's, a series of studies has showed that memory-borne dataflow is quite predictable: Franklin and Sohi's work in the context of Multiscalar [27], Moshovos and Sohi's [28] and Tyson and Austin's [29] simultaneously published studies on memory renaming, Chrysos and Emer's store sets [30], Sha, Martin, and Roth's Store Queue Index Prediction (SQIP) proposal [31], and others. From this we conclude that memory renaming is a promising approach to achieve equivalent dataflow tagging. Just as static solutions like branch hints were obviated by dynamic prediction, perhaps register encodings will eventually see a similar obsolescence.

But what should we expect of the greatly increased memory references that would normally be register operands? Here we gently note that register operands must be "well-behaved" by definition; if it was possible for the data to be aliased through memory, the compiler would be forced to not persist the value in a register in the first place. So in effect, by encoding such operands via (*e.g.*) stack offsets, we would only increase the overall predictability of memory dependencies, and most of these dependencies would be obvious through known methods.

We call the ultimate endpoint of this approach a *stateless ISA*, *i.e.* an ISA defined strictly in terms of state transitions in the system memory, with the computational device having no external state of its own. As such an architecture can use cache coherence to lay claim to a subset of memory and realize these transitions using the highest performance methods possible, the same aspect also directly implies that task switching is simply a matter of caching. This is promising for far more than just the killer microsecond problem. As we envision future heterogeneous systems out of necessity, the cost in time and energy of coordinating computations across various computational devices has a first order impact on the value derived from their differentiated attributes. The killer microsecond problem is just one simple example of how the overhead of context switching represents a sort of fundamental constant; lowering this constant will impact other aspects of system efficiency.

D. The hyper-taste

Our main thrust has been to argue that ISAs have an outmoded emphasis on microarchitectural concerns, noting that a great many microarchitectural techniques have been devised to transparently solve these problems to great effect. We should instead consider how to be more helpful to the consumers of our interface. Looking upwards in the system stack, the first immediate consumers of the architecture abstraction are hypervisors, virtual machine monitors, microkernels, and operating systems. How might we provide acceleration of their operations? What might an architecture with a taste of hypervisor consist of? How might we provide easy-to-use heterogeneous computation without forcing these software layers to continually adapt to our innovations?

VI. Conclusion

Many ideas proposed for the original MIPS architecture did not catch on, and we argue that overall, the seminal MIPS embodiment of the RISC philosophy provides many examples of what not to do in ISA design. However, it was not just the specific techniques that were flawed, but rather we posit that the real flaw is viewing hardware details exposed to the ISA as beneficial rather than, as we have argued, disadvantageous. From this perspective, practically all known ISAs have exposed implementation details to some degree, but the RISC philosophy's emphasis on exposing the microarchitecture in the name of performance sacrifices the future for the present.

Another perspective is that leaving a task to software is often a sound technical decision and conveniently allows further explorations on different solutions. Once a great deal of investment has been made in the software, however, it has proved difficult to change the interface despite potential advantages.

The lasting legacies of the RISC movement are simpler instructions intended to map directly onto hardware and load-store memory architecture. These attributes are appropriate for the microcode and μISA layer, but are not necessarily the right level of abstraction for a widely targetable ISA. We instead advocate an investigation into ISAs that say as little as possible, ideally nothing, about their implementation. This re-partitioning of responsibilities between hardware and software would provide not only maximum flexibility over time, but would allow more diversity of implementations, supporting our imminent and foreseeable needs for heterogeneous compute [32] in the post-Moore, post-Dennard era.

Acknowledgment

The authors would like to thank Reid McKenzie, Chad Wellington, Akanksha Jain, and the anonymous reviewers for their detailed and insightful feedback.

References

[1] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: a VLSI processor architecture," in *VLSI Systems and Computations*. Springer, 1981, pp. 337–346.

[2] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2003, pp. 15–24.

[3] A. Z. D. Herman, L. Wagner, and A. Zakai, "asm.js–working draft–18 august 2014," 2014.

[4] WebAssembly. [Online]. Available: webassembly.org

[5] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," 2015.

[6] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[7] W. Allcorn, *The Maginot Line 1928–45*. Bloomsbury Publishing, 2012.

[8] H. T. Datenschutz and D. Pataky, "Intel management engine," 2017.

[9] M. Ermolov and M. Goryachy, "How to hack a turned-off computer, or running unsigned code in intel management engine," in *Black Hat Europe 2017*.

[10] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *26th USENIX Security Symposium*, 2017.

[11] Gcc, the gnu compiler collection. [Online]. Available: gcc.gnu.org

[12] The llvm compiler infrastructure. [Online]. Available: llvm.org

[13] H. Corporaal, "Design of transport triggered architectures," in *Proceedings of 4th Great Lakes Symposium on VLSI*, 1994, pp. 130–135.

[14] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua *et al.*, "Baring it all to software: Raw machines," *Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[15] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "TRIPS: A polymorphous architecture for exploiting ILP, TLP, and DLP," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 1, no. 1, pp. 62–93, 2004.

[16] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[17] P. Suresh and R. Moona, "Perl-a registerless architecture," pp. 33–40, Dec 1998.

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization & Design: The Hardware/Software Interface, Fourth Edition*. Morgan Kaufmann Publishers, Inc.

[19] S. D. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. J. Sullivan, and T. Grutkowski, "The implementation of the Itanium 2 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1448–1460, 2002.

[20] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 98–105.

[21] G. J. Myers, "The case against stack-oriented instruction sets," *SIGARCH Comput. Archit. News*, 1977.

[22] 8-bit avr microcontroller with 16k bytes in-system programmable flash. http://www.atmel.com/images/doc2466.pdf. Accessed December 2017.

[23] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[24] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, 1962, pp. 335–344.

[25] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt, "How to fake 1000 registers," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005, pp. 7–18.

[26] D. W. Oehmke, *Virtualizing register context*. University of Michigan, 2005.

[27] M. Franklin and G. S. Sohi, "ARB: a hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, pp. 552–571, May 1996.

[28] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30, 1997, pp. 235–245.

[29] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30, 1997, pp. 218–227.

[30] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 142–153.

[31] T. Sha, M. M. K. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38, 2005, pp. 159–170.

[32] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 225–236.