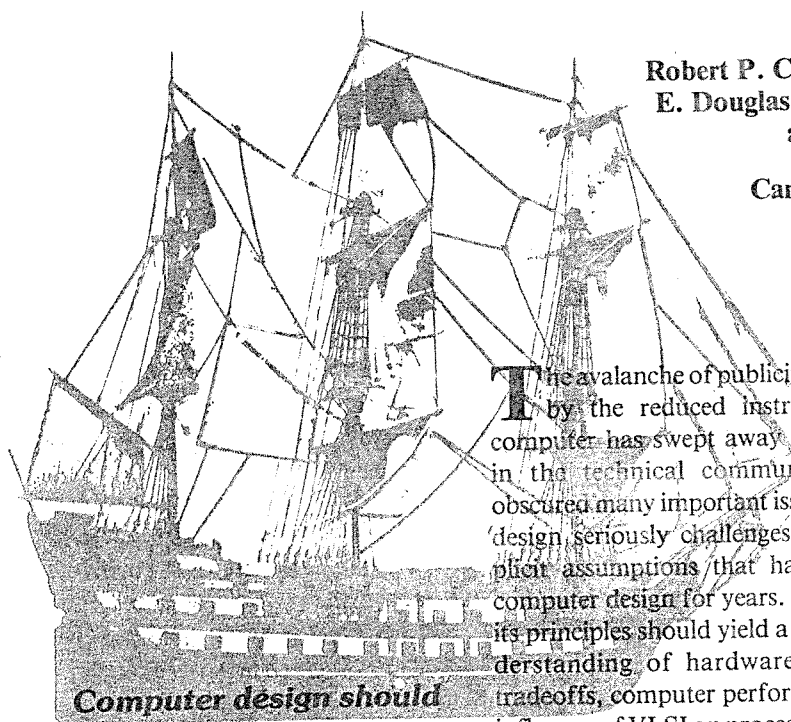


Instruction Sets and Beyond: Computers, Complexity, and Controversy

Robert P. Colwell, Charles Y. Hitchcock III,
E. Douglas Jensen, H. M. Brinkley Sprunt,
and Charles P. Kollar

Carnegie-Mellon University



Computer design should focus on the assignment of system functionality to implementation levels within an architecture, and not be guided by whether it is a RISC or CISC design.

The avalanche of publicity received by the reduced instruction set computer has swept away objectivity in the technical communities and obscured many important issues. RISC design seriously challenges some implicit assumptions that have guided computer design for years. A study of its principles should yield a deeper understanding of hardware/software tradeoffs, computer performance, the influence of VLSI on processor design, and many other topics. Articles on RISC research, however, often fail to explore these topics properly and can be misleading. Further, the few papers that present comparisons with complex instruction set computer design often do not address the same issues. As a result, even careful study of the literature is likely to give a distorted view of this area of research. This article offers a useful perspective of RISC/Complex Instruction Set Computer research, one that is supported by recent work at Carnegie-Mellon University.

Much RISC literature is devoted to discussions of the size and complexity of computer instruction sets. These discussions are extremely misleading.

Instruction set design is important, but it should not be driven solely by adherence to convictions about design style, RISC or CISC. The focus of discussion should be on the more general question of the assignment of system functionality to implementation levels within an architecture. This point of view encompasses the instruction set—CISCs tend to install functionality at lower system levels than RISCs—but also takes into account other design features such as register sets, coprocessors, and caches.

While the implications of RISC research extend beyond the instruction set, even within the instruction set domain, there are limitations that have not been identified. Typical RISC papers give few clues about where the RISC approach might break down. Claims are made for faster machines that are cheaper and easier to design and that “map” particularly well onto VLSI technology. It has been said, however, that “Every complex problem has a simple solution...and it is wrong.” RISC ideas are not “wrong,” but a simple-minded view of them would be. RISC theory has many implications that are not obvious. Re-

search in this area has helped focus attention on some important issues in computer architecture whose resolutions have too often been determined by defaults; yet RISC proponents often fail to discuss the application, architecture, and implementation contexts in which their assertions seem justified.

While RISC advocates have been vocal concerning their design methods and theories, CISC advocates have been disturbingly mute. This is not a healthy state of affairs. Without substantive, reported CISC research, many RISC arguments are left uncountered and, hence, out of perspective. The lack of such reports is due partially to the proprietary nature of most commercial CISC designs and partially to the fact that industry designers do not generally publish as much as academics. Also, the CISC design style has no coherent statement of design principles, and CISC designers do not appear to be actively working on one. This lack of a manifesto differentiates the CISC and RISC design styles and is the result of their different historical developments.

Towards defining a RISC

Since the earliest digital electronic computers, instruction sets have tended to grow larger and more complex. The 1948 MARK-1 had only seven instructions of minimal complexity, such as adds and simple jumps, but a contemporary machine like the VAX has hundreds of instructions. Furthermore, its instructions can be rather complicated, like atomically inserting an element into a doubly linked list or evaluating a floating point polynomial of arbitrary degree. Any high performance implementation of the VAX, as a result, has to rely on complex implementation techniques such as pipelining, prefetching, and multi-cycle instruction execution.

This progression from small and simple to large and complex instruction sets is striking in the development of single-chip processors within the past decade. Motorola's 68020, for example, carries 11 more addressing modes than the 6800, more than twice as many instructions, and support for an instruction cache and coprocessors. Again, not only has the number of addressing modes and instructions increased, but so has their complexity.

This general trend toward CISC machines was fueled by many things, including the following:

- New models are often required to be upward-compatible with existing models in the same computer family, resulting in the supersetting and proliferation of features.
- Many computer designers tried to reduce the "semantic gap" between programs and computer instruction sets. By adding instructions semantically closer to those used by programmers, these designers hoped to reduce software costs by creating a more easily programmed machine. Such instructions tend to be more complex because of their higher semantic level. (It is often the case, however, that instructions with high semantic content do not exactly match those required for the language at hand.)
- In striving to develop faster machines, designers constantly moved functions from software to microcode and from microcode to hardware, often without concern for the adverse effects that an added architectural feature can have on an implementation. For example, addition of an instruction requiring an extra level of decoding logic can slow a machine's entire instruction set. (This is called the " $n+1$ " phenomenon.¹)
- Tools and methodologies aid designers in handling the inherent

complexity of large architectures. Current CAD tools and microcoding support programs are examples.

Microcode is an interesting example of a technique that encourages complex designs in two ways. First, it provides a structured means of effectively creating and altering the algorithms that control execution of numerous operations and complex instructions in a computer. Second, the proliferation of CISC features is encouraged by the quantum nature of microcode memories; it is relatively easy to add another addressing mode or obscure instruction to a machine which has not yet used all of its microcode space.

Instruction traces from CISC machines consistently show that few of the available instructions are used in most computing environments. This situation led IBM's John Cocke, in the early 70's, to contemplate a departure from traditional computer styles. The result was a research project based on an ECL machine that used a very advanced compiler, creatively named "801" for the research group's building number. Little has been published about that project, but what has been released speaks for a principled and coherent research effort.

The 801's instruction set was based on three design principles. According to Radin,² the instruction set was to be that set of run-time operations that

- could not be moved to compile time,
- could not be more efficiently executed by object code produced by a compiler that understood the high-level intent of the program, and
- could be implemented in random logic more effectively than the equivalent sequence of software instructions.

The machine relied on a compiler that used many optimization strategies for much of its effectiveness, including a

powerful scheme of register allocation. The hardware implementation was guided by a desire for leanness and featured hardwired control and single-cycle instruction execution. The architecture was a 32-bit load/store machine (only load and store instructions accessed memory) with 32 registers and single-cycle instructions. It had separate instruction and data caches to allow simultaneous access to code and operands.

Some of the basic ideas from the 801 research reached the West Coast in the mid 70's. At the University of California at Berkeley, these ideas grew into a series of graduate courses that produced the RISC I* (followed later by the RISC II) and the numerous CAD tools that facilitated its design. These courses laid the foundation for related research efforts in performance evaluation, computer-aided design, and computer implementation.

The RISC I processor,³ like the 801, is a load/store machine that executes most of its instructions in a single cycle. It has only 31 instructions, each of which fits in a single 32-bit word and uses practically the same encoding format. A special feature of the RISC I is its large number of registers, well over a hundred, which are used to form a series of overlapping register sets. This feature makes procedure calls on the RISC I less expensive in terms of processor-memory bus traffic.

Soon after the first RISC I project at Berkeley, a processor named MIPS (Microprocessor without Interlocked Pipe Stages) took shape at Stanford. MIPS¹ is a pipelined, single-chip processor that relies on innovative software to ensure that its pipeline resources are properly managed. (In machines such as the IBM System/360 Model 91, pipeline interstage interlocking is per-

formed at run-time by special hardware). By trading hardware for compile-time software, the Stanford researchers were able to expose and use the inherent internal parallelism of their fast computing engine.

These three machines, the 801, RISC I, and MIPS, form the core of RISC research machines, and share a set of common features. We propose the following elements as a working definition of a RISC:

- (1) *Single-cycle operation* facilitates the rapid execution of simple functions that dominate a computer's instruction stream and promotes a low interpretive overhead.
- (2) *Load/store design* follows from a desire for single-cycle operation.
- (3) *Hardwired control* provides for the fastest possible single-cycle operation. Microcode leads to slower control paths and adds to interpretive overhead.
- (4) *Relatively few instructions and addressing modes* facilitate a fast, simple interpretation by the control engine.
- (5) *Fixed instruction format* with consistent use, eases the hardwired decoding of instructions, which again speeds control paths.
- (6) *More compile-time effort* offers an opportunity to explicitly move static run-time complexity into the compiler. A good example of this is the software pipeline reorganizer used by MIPS.¹

A consideration of the two companies that claim to have created the first commercial "RISC" computer, Ridge Computers and Pyramid Technology, illustrates why a definition is needed. Machines of each firm have restricted instruction formats, a feature they share with RISC machines.

Pyramid's machine is not a load/store computer, however, and both Ridge and Pyramid machines have variable length instructions and use multiple-cycle interpretation and microcoded control engines. Further, while their instruction counts might seem reduced when compared to a VAX, the Pyramid has almost 90 instructions and the Ridge has over 100. The use of microcoding in these machines is for price and performance reasons. The Pyramid machine also has a system of multiple register sets derived from the Berkeley RISC I, but this feature is orthogonal to RISC theory. These may be successful machines, from both technological and marketing standpoints, but they are not RISCs.

The six RISC features enumerated above can be used to weed out misleading claims and provide a springboard for points of debate. Although some aspects of this list may be arguable, it is useful as a working definition.

Points of attention and contention

There are two prevalent misconceptions about RISC and CISC. The first is due to the RISC and CISC acronyms, which seem to imply that the domain for discussion should be restricted to selecting candidates for a machine's instruction set. Although specification format and number of instructions are the primary issues in most RISC literature, the best generalization of RISC theory goes well beyond them. It connotes a willingness to make design tradeoffs freely and consciously across architecture/implementation, hardware/software, and compile-time/run-time boundaries in order to maximize performance as measured in some specific context.

The RISC and CISC acronyms also seem to imply that any machine can be classified as one or the other and that

* Please note that the term "RISC" is used throughout this article to refer to all research efforts concerning Reduced Instruction Set Computers, while the term "RISC I" refers specifically to the Berkeley research project.

the primary task confronting an architect is to choose the most appropriate design style for a particular application. But the classification is not a dichotomy. RISCs and CISCs are at different corners of a continuous multidimensional design space. The need is not for an algorithm by which one can be chosen; rather, the goal should be the formulation of a set of techniques, drawn from CISC experiences and RISC tenets, which can be used by a designer in creating new systems.⁴⁻⁶

One consequence of the us-or-them attitude evinced by most RISC publications is that the reported performance of a particular machine (e.g., RISC I) can be hard to interpret if the contributions made by the various design decisions are not presented individually. A designer faced with a large array of choices needs guidance more specific than a monolithic, all-or-nothing performance measurement.

An example of how the issue of scope can be confused is found in a recent article.⁷ By creating a machine with only one instruction, its authors claim to have delimited the RISC design space to their machine at one end of the space and the RISC I (with 31 instructions) at the other end. This model is far too simplistic to be useful; an absolute number of instructions cannot be the sole criterion for categorizing an architecture as to RISC or CISC. It ignores aspects of addressing modes and their associated complexity, fails to deal with compiler/architecture coupling, and provides no way to evaluate the implementation of other non-instruction set design decisions such as register files, caches, memory management, floating point operations, and co-processors.

Another fallacy is that the total system is composed of hardware, software, and application code. This leaves out the operating system, and the overhead and the needs of the operating system cannot be ignored in most systems. This area has received

far too little attention from RISC research efforts, in contrast to the CISC efforts focused on this area.^{8,9}

An early argument in favor of RISC design was that simpler designs could be realized more quickly, giving them a performance advantage over complex machines. In addition to the economic advantages of getting to market first, the simple design was supposed to

The insinuation that the MicroVAX-32 follows in a RISC tradition is unreasonable. It does not follow our definition of a RISC; it violates all six RISC criteria.

avoid the performance disadvantages of introducing a new machine based on relatively old implementation technology. In light of these arguments, DEC's MicroVAX-32¹⁰ is especially interesting.

The VAX easily qualifies as a CISC. According to published reports, the MicroVAX-32, a VLSI implementation of the preponderance of the VAX instruction set, was designed, realized, and tested in a period of several months. One might speculate that this very short gestation period was made possible in large part by DEC's considerable expertise in implementing the VAX architecture (existing products included the 11/780, 11/750, 11/730, and VLSI-VAX). This shortened design time would not have been possible had DEC had not first created a standard instruction set. Standardization at this level, however, is precisely what RISC theory argues against. Such standards constrain the unconventional RISC hardware/software tradeoffs. From a commercial standpoint, it is significant that the MicroVAX-32 was born into a world where compatible assemblers, compilers, and operating systems abound, something that would certainly not be the case for a RISC design.

Such problems with RISC system designs may encourage commercial RISC designers to define a new level of standardization in order to achieve some of the advantages of multiple implementations supporting one standard interface. A possible choice for such an interface would be to define an intermediate language as the target for all compilation. The intermediate language would then be translated into optimal machine code for each implementation. This translation process would simply be performing resource scheduling at a very low level (e.g., pipeline management and register allocation).

It should be noted that the MicroVAX-32 does not directly implement all VAX architecture. The suggestion has been made that this implementation somehow supports the RISC inclination toward emulating complex functions in software. In a recent publication, David Patterson observed:

Although I doubt DEC is calling them RISCs, I certainly found it interesting that DEC's single chip VAXs do not implement the whole VAX instruction set. A MicroVAX traps when it tries to execute some infrequent but complicated operations, and invokes transparent software routines that simulate those complicated instructions.¹¹

The insinuation that the MicroVAX-32 follows in a RISC tradition is unreasonable. It does not come close to fitting our definition of a RISC; it violates all six RISC criteria. To begin with, any VAX by definition has a variable-length instruction format and is not a load/store machine. Further, the MicroVAX-32 has multicycle instruction execution, relies on a micro-coded control engine, and interprets the whole array of VAX addressing modes. Finally, the MicroVAX-32 executes 175 instructions on-chip, hardly a reduced number.

A better perspective in the MicroVAX-32 shows that there are indeed cost/performance ranges where microcoded implementation of certain functions is inappropriate and software emulation is better. The importance of carefully making this assignment of function to implementation level—software, microcode, or hardware—has been amply demonstrated in many RISC papers. Yet this basic concern is also evidenced in many CISC machines. In the case of the MicroVAX-32, floating point instructions are migrated either to a coprocessor chip or to software emulation routines. The numerous floating-point chips currently available attest to the market reception for this partitioning. Also migrated to emulation are the console, decimal, and string instructions. Since many of these instructions are infrequent, not time-critical, or are not generated by many compilers, it

would be difficult to fault this approach to the design of an inexpensive VAX. The MicroVAX-32 also shows that it is still possible for intelligent, competent computer designers who understand the notion of correct function-to-level mapping to find microcoding a valuable technique. Published RISC work, however, does not accommodate this possibility.

The application environment is also of crucial importance in system design. The RISC I instruction set was designed specifically to run the C language efficiently, and it appears reasonably successful. The RISC I researchers have also investigated the Smalltalk-80 computing environment.¹² Rather than evaluate RISC I as a Smalltalk engine, however, the RISC I researchers designed a new RISC and report encouraging performance results from simulations. Still, designing a processor to run a single

language well is different from creating a single machine such as the VAX that must exhibit at least acceptable performance for a wide range of languages. While RISC research offers valuable insights on a per-language basis, more emphasis on cross-language anomalies, commonalities, and tradeoffs is badly needed.

Especially misleading are RISC claims concerning the amount of design time saved by creating a simple machine instead of a complex one. Such claims sound reasonable. Nevertheless, there are substantial differences in the design environments for an academic one-of-a-kind project (such as MIPS or RISC I) and a machine with lifetime measured in years that will require substantial software and support investments. As was pointed out in a recent *Electronics Week* article, R. D. Lowry, market development manager for Denelcor,

Risc II and the MCF evaluation

In the mid 70's, a committee was created by the Department of Defense to "evaluate the efficiency of several computer architectures independently of their implementations."^{1,2} This committee developed the Military Computer Family studies based on the premise that the "architectural efficiency" of a computer corresponds to its life-cycle cost, given some standard of implementation technology. The MCF committee developed a means of evaluating architectural efficiency that consisted of two parts: (1) an initial screening to determine the "reasonableness" of an architecture based on several qualitative and quantitative factors (described later) and (2) a methodical application of benchmarks for machines that successfully passed this screening.

The MCF evaluations have been considered by many to be an important milestone in the systematic evaluation of computer architec-

tures. The published evaluations of RISC machines have indicated performance advantages large enough to merit attention and analysis. To learn about RISC architecture and the usefulness of the MCF evaluation procedure, we applied the complete MCF evaluation to the Berkeley RISC II since it posed the fewest obstacles.

The MCF program evaluates architectures standardized at the instruction set level, since, according to Burr, it "is the only [way to ensure] complete software transportability across a wide range of computer implementations."¹ This view is contrary to a fundamental RISC tenet that one should zealously pursue unconventional tradeoffs across the architecture/implementation boundary that can produce higher performance.

In addition, the architecture that was judged the best by the MCF evaluation criteria was the VAX, a particularly intriguing judgement considering the uniformly bad reviews

given the VAX in RISC performance studies. Furthermore many of these RISC performance studies used variations and carefully chosen subsets of the MCF benchmarks.³ Evaluating the RISC II with a full MCF analysis sheds new light on this seeming discrepancy.

MCF evaluation criteria. The first part of the MCF evaluation is an initial screening to ensure that the candidate architecture contains features deemed essential to a successful military computer: virtual memory, protection, floating point, interrupts and traps, subsetability, multiprocessor support, I/O controllability, extensibility, and the ability to execute out of read-only memory. Current RISC II systems have not provided many of these features, but most of these requirements could be met with additional resources.

The initial screening also analyzes quantitative factors. Since this

noted that "commercial-product development teams generally start off a project by weighing the profit and loss impacts of design decisions."¹³ Lowry is quoted as saying, "A university doesn't have to worry about that, so there are often many built-in deadends in projects. This is not to say the value of their research is diminished. It does, however, make it very difficult for someone to reinvent the system to make it a commercial product." For a product to remain viable, a great deal of documentation, user training, coordination with fabrication or production facilities, and future upgrades must all be provided. It is not known how these factors might skew a design-time comparison, so all such comparisons should be viewed with suspicion.

Even performance claims, perhaps the most interesting of all RISC assertions, are ambiguous. Performance as measured by narrowly compute-

bound, low-level benchmarks that have been used by RISC researchers (e.g., calculating a Fibonacci series recursively) is not the only metric in a computer system. In some, it is not even one of the most interesting. For many current computers, the only useful performance index is the number of transactions per second, which has no direct or simple correlation to the time it takes to calculate Ackermann's function. While millions of instructions per second might be a meaningful metric in some computing environments, reliability, availability, and response time are of much more concern in others, such as *space* and *aviation* computing. The extensive error checking incorporated into these machines at every level may slow the basic clock time and substantially diminish performance. Reduced performance is tolerable; but downtime may not be. In the extreme, naive application of

the RISC rules for designing an instruction set might result in a missile guidance computer optimized for running its most common task—diagnostics. In terms of instruction frequencies, of course, flight control applications constitute a trivial special case and would not be given much attention. It is worth emphasizing that in efforts to quantify performance and apply those measurements to system design, one must pay attention not just to instruction execution frequencies, but also to cycles consumed per instruction execution. Levy and Clark make this point regarding the VAX instruction set,¹⁴ but it has yet to appear in any papers on RISC.

When performance, such as throughput or transactions per second, is a first-order concern, one is faced with the task of quantifying it. The Berkeley RISC I efforts to establish the machine's throughput are laudable, but

screening includes such practicalities as the manufacturer's current customer base and the amount of existing software, the RISC II would compare unfavorably to the VAX in this part of the evaluation. While these factors were important in military computer standards, they are clearly irrelevant here.

After the initial screening, a series of test programs was executed on a simulator of the candidate architecture. To avoid compiler ambiguities, the benchmarks were programmed in the assembly language of the test system. The MCF committee was interested solely in compiled code performance, yet the members recognized that varying levels of compiler technology should not be allowed to affect the outcome of the study; compiler sophistication has nothing to do with inherent "architectural efficiency." At the time of the MCF evaluations, it was believed that even the best compilers would be unlikely to

generate better code than expert programmers. Sixteen benchmark programs were developed: they were representative of the tasks performed by military computers and were small enough for humans to write in a highly optimized form.

None of the sixteen benchmarks tests methods of subroutine linkage (although one of the benchmarks considered, but rejected, for the MCF study was the highly recursive Ackermann's function). Failure to test call efficiency was not an oversight by the MCF committee; two measures of subroutine efficiency are included in the quantitative factors section of the initial screening.

Rather than rely on combined architecture/implementation measurements such as execution throughput, the MCF measures of computer architecture efficiency were defined to be program size (S), memory bus traffic (M), and canonical processor cycles (R). The S measure includes

the local data and stack space used by the benchmark, as well as its program space. (The benchmarks reflect a circa-1970 assumption that code and data each occupy about half of the available memory space.) The M measure for a benchmark is the number of bytes that the processor reads and writes to memory (no transparent caching scheme is used). To compute the R measure, the architecture being evaluated is emulated on a canonical processor. The R measure is the sum of the internal data register-to-register transfers required by the canonical processor. Thus, this measure is supposed to model the data traffic of the processor's internal activities during benchmark execution. To evaluate different architectures, these measures are used as dimensions of comparison.

RISC theory asserts that simple instructions can be made to execute very quickly if their implementations are unencumbered by the large con-

before sweeping conclusions are drawn one must carefully examine the benchmark programs used. As Patterson noted:

The performance predictions for [RISC I and RISC II] were based on small programs. This small size was dictated by the reliability of the simulator and compiler, the available simulation time, and the inability of the first simulators to handle UNIX system calls.¹¹

Some of these "small" programs actually execute millions of instructions, yet they are very narrow programs in terms of the scope of function. For example, the Towers of Hanoi program, when executing on the 68000, spends over 90 percent of its memory accesses in procedure calls and returns. The RISC I and II researchers recently reported results from a large benchmark,¹¹ but the importance of large,

heterogenous benchmarks in performance measurement is still lost on many commercial and academic computer evaluators who have succumbed to the misconception that "micro-benchmarks" represent a useful measurement in isolation.

Multiple register sets

Probably the most publicized RISC-style processor is the Berkeley RISC I. The best-known feature of this chip is its large register file, organized as a series of overlapping register sets. This is ironic, since the register file is a performance feature independent of any RISC (as defined earlier) aspect of the processor. Multiple register sets could be included in any general-purpose register machine.

It is easy to believe that MRSs can yield performance benefits, since procedure-based, high-level languages

typically use registers for information specific to a procedure. When a procedure call is performed, the information must be saved, usually on a memory stack, and restored on a procedure return. These operations are typically very time consuming due to the intrinsic data transfer requirements. RISC I uses its multiple register sets to reduce the frequency of this register saving and restoring. It also takes advantage of an overlap between register sets for parameter passing, reducing even further the memory reads and writes necessary.¹⁵

RISC I has a register file of 138 32-bit registers organized into eight overlapping "windows." In each window, six registers overlap the next window (for outgoing parameters and incoming results). During any procedure, only one of these windows is actually accessible. A procedure call changes the current window to the next

control engine normally required for complex instructions. Consequently, since the MCF evaluation avoids measuring implementation features, any performance gains realized by such simplified control engines are ignored, while penalties, such as the increased processor-memory traffic of these load/store machines, are still taken into account. This effect has been noted before in applying the MCF evaluation to real machines.⁴

Results and interpretation. The RISC II architecture was evaluated by simulating assembly language versions of the 16 benchmarks. To gauge the results, its performance was compared to that of the VAX, rated "best" by the MCF measures. The VAX had a significantly lower S measure (memory space requirements) in 14 of the 16 benchmarks, requiring an average of three and a half times less memory than RISC II. This result seems inconsistent with published RISC reports

which found that the RISC I took an average of only 50 percent more memory. This difference is dramatic. Much of it may be due to the fact that previous studies used a compiler that produces reasonable code for the RISC II, but produces suboptimal code for the VAX (since it may not have been sophisticated enough to exploit the available complex instructions as a human would). If the latest compilers were used for both machines, the space difference between the machines would likely be reduced to that of handcoding in assembly language, which we used.

The RISC II had a much higher M (memory traffic measure) for 11 of the benchmarks, averaging over two and a half times more processor-memory traffic than the VAX. This MCF criterion shows the large penalty paid by RISC II because of its load/store architecture. It is accentuated by the generic RISC need to fetch more instructions per program,

since RISC instructions have low semantic content.

The VAX also had a lower (better) R measure for 10 of the benchmarks, and it was substantially lower on five of them. Again, much of this difference was due to the increased number of instruction fetches required by RISCs. One of the ten benchmarks modelled the cost of a context swap, which is high on RISC II because of the amount of state information in the register file. On average, about half of the register file (approximately 64 registers) must be saved and restored in each process swap.

These benchmarks showed the RISC II to disadvantage on floating point, integer multiplication, bit test and set operations, variable-sized block moves, and character string searches—operations for which RISC II has no primitive instructions. As a result, numerous instructions are required to emulate on a RISC

window by incrementing a pointer, and the six outgoing parameter registers become the incoming parameters of the called procedure. Similarly, a procedure return changes the current window to the previous window, and the outgoing result registers become the incoming result registers of the calling procedure. If we assume that six 32-bit registers are enough to contain the parameters, a procedure call involves no actual movement of information (only the window pointer is adjusted). The finite on-chip resources limit the actual savings due to register window overflows and underflows.³

It has been claimed that the small control area needed to implement the simple instruction set of a VLSI RISC leaves enough chip area for the large register file.³ The relatively small amount of control logic used by a RISC does free resources for other uses, but a large register file is not the

only way to use them, nor even necessarily the best. For example, designers of the 801 and MIPS chose other ways to use their available hardware; these RISCs have only a single, conventionally sized register set. Caches, floating-point hardware, and interprocess communication support are a few of the many possible uses for those resources "freed" by a RISC's simple instruction set. Moreover, as chip technology improves, the tradeoffs between instruction set complexity and architecture/implementation features become less constrained. Computer designers will always have to decide how to best use available resources and, in doing so, should realize which relations are intrinsic and which are not.

The Berkeley papers describing the RISC I and RISC II processors claimed their resource decisions produced large performance improvements, two to four times over CISC machines like

the VAX and the 68000.^{3,11} There are many problems with these results and the methods used to obtain them. Foremost, the performance effects of the reduced instruction set were not decoupled from those of the overlapped register windows. Consequently, these reports shed little light on the RISC-related performance of the machine, as shown below.

Some performance comparisons between different machines, especially early ones, were based on simulated benchmark execution times. While absolute speed is always interesting, other metrics less implementation-dependent can provide design information more useful to computer architects, such as data concerning the processor-memory traffic necessary to execute a series of benchmarks. It is difficult to draw firm conclusions from comparisons of vastly different machines unless some effort has been

what other machines provide in their instruction set; the MCF study provides a quantitative evaluation of this effect. The RISC II was comparable to the VAX on benchmarks that involved simple arithmetic and one-level array indexing.

Conclusions. Although the VAX achieves a better score on every aspect of the MCF evaluation than does the RISC II, it would be dangerous to conclude that the VAX is a "better" machine. The MCF study characterizes the life-cycle costs of various architectures based on a set of weighting factors culled from the military environment. The VAX can be said to be better only in the sense that the MCF life-cycle cost models clearly favor it.

Since RISC research explicitly gives up the possible benefits of the traditional architecture/implementation dichotomy to increase execution throughput, the most basic MCF

tenet does not hold for RISCs. The MCF life-cycle cost models did not include execution throughput, so the RISC II performance-related features were ignored, yet the price paid for these features is clear.

The MCF study's S.M. and R measures of architectural efficiency are open to question. For example, the R measure of internal processor overhead is of dubious utility when the architectures being compared are dissimilar. It is hard to see what canonical processor could be devised to serve as a common implementation of the RISC II and the Intel 432, for example.

The MCF study remains, however, the only large-scale evaluation of computer systems that includes as a primary figure of merit system life-cycle costs instead of easy throughput comparisons based on many arbitrary and implicit assumptions. The care taken by MCF in factoring out the myriad interrelated elements of a

computer system leaves it an excellent model for future evaluation efforts.

References

1. W.E. Burr, A.H. Coleman, and W.R. Smith, "Overview of the Military Computer Family Architecture Selection," *NCC Conference Proceedings AFIPS*, Montvale, N.J., 1977, pp. 131-137.
2. S.H. Fuller and W.E. Burr, "Measurement and Evaluation of Alternative Computer Architectures," *Computer*, Vol. 10, No. 10, Oct. 1977, pp. 24-35.
3. David A. Patterson, Richard S. Piepho, "RISC Assessment: A High-Level Language Experiment," *Proc. Ninth Ann. Symp. Computer Architecture*, 1982, pp. 3-8.
4. J.B. Mountain and P.H. Enslow Jr., "Application of the Military Computer Family Architecture Selection Criteria to the PRIME P400," *Computer Architecture News*, Vol. 6, No. 6, Feb. 1978.
5. D. Patterson, "RISC Watch," *Computer Architecture News*, Vol. 12, No. 1, Mar. 1984, pp. 11-19.

made to factor out implementation-dependent features not being compared (e.g., caches and floating point accelerators).

Experiments structured to accommodate these reservations were conducted at CMU to test the hypothesis that the effects of multiple register sets are orthogonal to instruction set complexity.¹⁶ Specifically, the goal was to see if the performance effects of MRSs were comparable for RISCs and CISCs. Simulators were written for two CISCs (the VAX and the 68000) without MRSs, with non-overlapping MRSs and with overlapping MRSs. Simulators were also written for the RISC I, RISC I with non-overlapping register sets, and RISC I with only a single register set. In each of the simulators, care was taken not to change the initial architectures any more than absolutely necessary to add or remove MRSs. Instead of simulating execution time, the total amount of processor-memory traffic (bytes read and written) for each benchmark was recorded for comparison. To use this data fairly, only different register set versions of the same architecture were compared so the ambiguities that arise from comparing different architectures like the RISC I and the VAX were avoided. The benchmarks used were the same ones originally used to evaluate RISC I. A summary of the experiments and their results are presented by Hitchcock and Sprunt.¹⁷

As expected, the results show a substantial difference in processor-memory traffic for an architecture with and without MRSs. The MRS versions of both the VAX and 68000 show marked decreases in processor-memory traffic for procedure-intensive benchmarks, shown in Figures 1 and 2. Similarly, the single register set version of RISC I requires many more memory reads and writes than RISC I with overlapped register sets (Figure 3). This result is due in part to the method used for

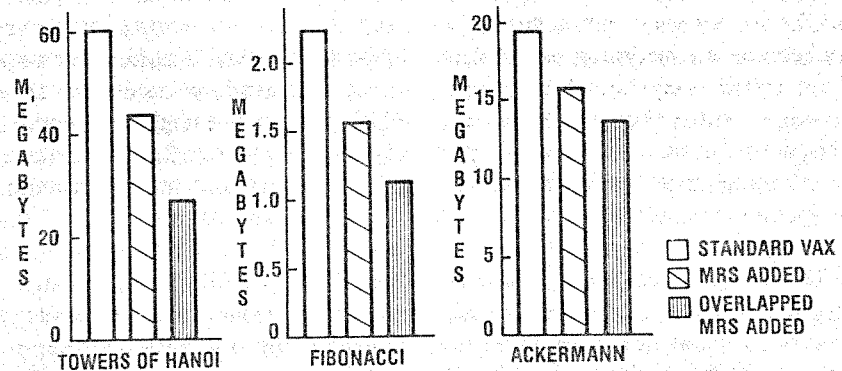


Figure 1. Total processor-memory traffic for benchmarks on the standard VAX and two modified VAX computers, one with multiple register sets and one with overlapped multiple register sets.

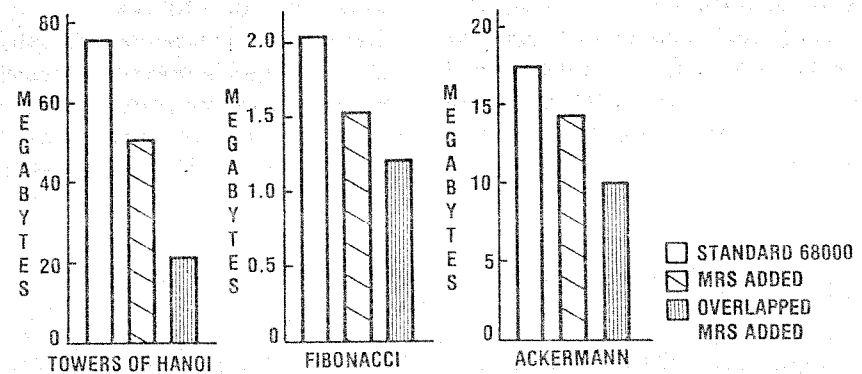


Figure 2. Total processor-memory traffic for benchmarks on the standard 68000 and two modified 68000s, one with multiple register sets and one with overlapped multiple register sets.

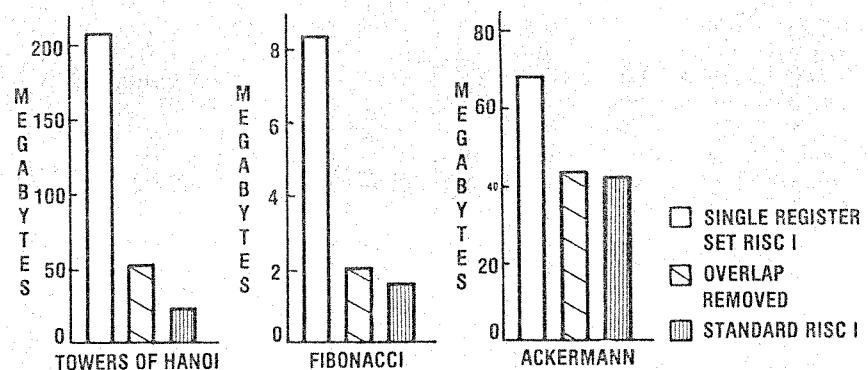


Figure 3. Total processor-memory traffic for benchmarks on the standard RISC I and two modified RISC I's, one with no overlap between register sets and one with only one register set.

handling register set overflow and underflow, which was kept the same for all three variations. With a more intelligent scheme, the single register set RISC I actually required fewer bytes of memory traffic on Ackermann's function than its multiple register set counterparts. For benchmarks with very few procedure calls (e.g., the sieve of Eratosthenes), the single register set version has the same amount of processor-memory traffic as the MRS version of the same architecture.¹⁷

Clearly, MRSs can affect the amount of processor-memory traffic necessary to execute a program. A significant amount of the performance of RISC I for procedure-intensive environments has been shown to be attributable to its scheme of overlapped register sets, a feature independent of instruction-set complexity. Thus, any performance claims for reduced instruction set computers that do not remove effects due to multiple register sets are inconclusive, at best.

These CMU experiments used benchmarks drawn from other RISC research efforts for the sake of continuity and consistency. Some of the benchmarks, such as Ackermann, Fibonacci, and Hanoi, actually spend most of their time performing procedure calls. The percentage of the total processor-memory traffic due to "C" procedure calls for these three benchmarks on the single register set version of the 68000 ranges from 66 to 92 percent. As was expected, RISC I, with its overlapped register structure that allows procedure calls to be almost free in terms of processor-memory bus traffic, did extremely well on these highly recursive benchmarks when compared to machines with only a single register set. It has not been established, however, that these benchmarks are representative of any computing environment.

The 432

The Intel 432 is a classic example of a CISC. It is an object-oriented VLSI microprocessor chip-set designed expressly to provide a productive Ada programming environment for large scale, multiple-process, multiple-processor systems. Its architecture supports object orientation such that every object is protected uniformly without regard to traditional distinctions such as "supervisor/user mode" or "system/user data structures." The 432 has a very complex instruction set. Its instructions are bit-encoded and range in length from six to 321 bits. The 432 incorporates a significant degree of functional migration from software to on-chip microcode. The interprocess communication SEND primitive is a 432 machine instruction, for instance.

Published studies of the performance of the Intel 432 on low-level benchmarks (e.g., towers of Hanoi¹⁸) show that it is very slow, taking 10 to 20 times as long as the VAX 11/780. Such a design, then, invites scrutiny in the RISC/CISC controversy.

One is tempted to blame the machine's object-oriented runtime environment for imposing too much overhead. Every memory reference is checked to ensure that it lies within the boundaries of the referenced object, and the read/write protocols of the executing context are verified. RISC proponents argue that the complexity of the 432 architecture, and the additional decoding required for a bit-encoded instruction stream contribute to its poor performance. To address these and other issues, a detailed study of the 432 was undertaken to evaluate the effectiveness of the architectural mechanisms provided in support of its intended runtime environment. The study concentrated on one of the central differences in the RISC and CISC design styles: RISC designs avoid hardware/microcode structures in-

tended to support the runtime environment, attempting instead to place equivalent functionality into the compiler or software. This is contrary to the mainstream of instruction set design, which reflects a steady migration of such functionality from higher levels (software) to lower ones (microcode or hardware) in the expectation of improved performance.

This investigation should include an analysis of the 432's efficiency in executing large-system code, since executing such code well was the primary design goal of the 432. Investigators used the Intel 432 microsimulator, which yields cycle-by-cycle traces of the machine's execution. While this microsimulator is well-suited to simulating small programs, it is quite unwieldy for large ones. As a result, the concentration here is on the low-level benchmarks that first pointed out the poor 432 performance.

Simulations of these benchmarks revealed several performance problems with the 432 and its compiler:

(1) The 432's Ada compiler performs almost no optimization. The machine is frequently forced to make unnecessary changes to its complex addressing environment, and it often recomputes costly, redundant subexpressions. This recomputation seriously skews many results from benchmark comparisons. Such benchmarks reflect the performance of the present version of the 432 but show very little about the efficacy of the architectural trade-offs made in that machine.

(2) The bandwidth of 432 memory is limited by several factors. The 432 has no on-chip data caching, no instruction stream literals, and no local data registers. Consequently, it makes far more memory references than it would otherwise have to. These reference requirements also make the code size much larger, since many more bits are required to reference data within an object than within a local register.

And because of pin limitations, the 432 must multiplex both data and address information over only 16 pins. Also, the standard Intel 432/600 development system, which supports shared-memory multiprocessing, uses a slow asynchronous bus that was designed more for reliability than throughput. These implementation factors combine to make wait states consume 25 to 40 percent of the processor's time on the benchmarks.

(3) On highly recursive benchmarks, the object-oriented overhead in the 432 does indeed appear in the form of a slow procedure call. Even here, though, the performance problems should not be attributed to object orientation or to the machine's intrinsic complexity. Designers of the 432 made a decision to provide a new, protected context for every procedure call; the user has no option in this respect. If an unprotected call mechanism were used where appropriate, the Dhrystone benchmark¹⁹ would run 20 percent faster.

(4) Instructions are bit-aligned, so the 432 must almost of necessity decode the various fields of an instruction sequentially. Since such decoding often overlaps with instruction execution, the 432 stalls three percent of the time while waiting for the instruction decoder. This percentage will get worse, however, once other problems above are eliminated.

Colwell provides a detailed treatment of this experiment and its results.²⁰

This 432 experiment is evidence that RISC's renewed emphasis on the importance of fast instruction decoding and fast local storage (such as caches or registers) is substantiated, at least for low-level compute-bound benchmarks. Still, the 432 does not provide compelling evidence that large-scale migration of function to microcode and hardware is ineffective. On the contrary, Cox et al.²¹ demonstrated

that the 432 microcode implementation of interprocess communication is much faster than an equivalent software version. On these low-level benchmarks, the 432 could have much higher performance with only a better compiler and minor changes to its implementation. Thus, it is wrong to conclude that the 432 supports the general RISC point of view.

In spite of—and sometimes because of—the wide publicity given to current RISC and CISC research, it is not easy to gain a thorough appreciation of the important issues. Articles on RISC research are often oversimplified, overstated, and misleading, and papers on CISC design offer no coherent design principles for comparison. RISC/CISC issues are best considered in light of their function-to-implementation level assignment. Strictly limiting the focus to instruction counts or other oversimplifications can be misleading or meaningless.

Some of the more subtle issues have not been brought out in current literature. Many of these are design considerations that do not lend themselves to the benchmark level analysis used in RISC research. Nor are they always properly evaluated by CISC designers, guided so frequently by tradition and corporate economics.

RISC/CISC research has a great deal to offer computer designers. These contributions must not be lost due to an illusory and artificial dichotomy. Lessons learned studying RISC machines are not incompatible with or mutually exclusive of the rich tradition of computer design that preceded them. Treating RISC ideas as perspectives and techniques rather than dogma and understanding their domains of applicability can add important new tools to a computer designer's repertoire. □

Acknowledgements

We would like to thank the innumerable individuals, from industry and academia, who have shared their thoughts on this matter with us and stimulated many of our ideas. In particular, we are grateful to George Cox and Konrad Lai of Intel for their help with the 432 microsimulator.

This research was sponsored in part by the Department of the Army under contract DAA B07-82-C-J164.

References

1. J. Hennessy et al., "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, 1982, pp. 2-11.
2. G. Radin, "The 801 Minicomputer," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, 1982, pp. 39-47.
3. D. A. Patterson and C. H. Sequin, "A VLSI RISC," *Computer*, Vol. 15, No. 9, Sept. 1982, pp. 8-21.
4. R. P. Colwell, C. Y. Hitchcock III, and E. D. Jensen, "A Perspective on the Processor Complexity Controversy," *Proc. Int. Conf. Computer Design: VLSI in Computers*, 1983, pp. 613-616.
5. D. Hammerstrom, "Tutorial: The Migration of Function into Silicon," *10th Ann. Int'l Symp. Computer Architecture*, 1983.
6. J. C. Browne, "Understanding Execution Behavior of Software Systems," *Computer*, Vol. 17, No. 7, July 1984, pp. 83-87.
7. H. Azaria and D. Tabak, "The MODHEL Microcomputer for RISCs Study," *Microprocessing and Microprogramming*, Vol. 12, No. 3-4, Oct.-Nov. 1983, pp. 199-206.
8. G. C. Barton "Sentry: A Novel Hardware Implementation of Classic Operating System Mechanisms," *Proc. Ninth Ann. Int'l Symp. Computer Architecture*, 1982, pp. 140-147.
9. A. D. Berenbaum, M. W. Condry, and P. M. Lu, "The Operating System and Language Support Features of the BELLMAC-32 Microprocessor,"

Proc. Symp. Architectural Support for Programming Languages and Operating Systems, 1982, pp. 30-38.

10. J. Hennessy, "VLSI Processor Architecture," *IEEE Transactions on Computers*, Vol. C-33, No. 12, Dec. 1984, pp. 1221-1246.
11. D. Patterson, "RISC Watch," *Computer Architecture News*, Vol. 12, No. 1, Mar. 1984, pp. 11-19.
12. David Ungar et al., "Architecture of SOAR: Smalltalk on a RISC," *11th Ann. Int'l Symp. Computer Architecture*, 1984, pp. 188-197.
13. W. R. Iversen, "Money Starting to Flow As Parallel Processing Gets Hot," *Electronics Week*, Apr. 22, 1985, pp. 36-38.
14. H. M. Levy and D. W. Clark, "On the Use of Benchmarks for Measuring System Performance" *Computer Architecture News*, Vol. 10, No. 6, 1982, pp. 5-8.
15. D. C. Halbert and P. B. Kessler, "Windows of Overlapping Register Frames", CS292R Final Reports, University of California, Berkeley, June 9, 1980.
16. R. P. Colwell, C. Y. Hitchcock III, and E. D. Jensen, "Peering Through the RISC/CISC Fog: An Outline of Research," *Computer Architecture News*, Vol. 11, No. 1, Mar. 1983, pp. 44-50.
17. C. Y. Hitchcock III and H. M. B. Sprunt, "Analyzing Multiple Register Sets," *12th Ann. Int'l Symp. Computer Architecture*, 1985, in press.
18. P. M. Hansen et al., "A Performance Evaluation of the Intel iAPX 432," *Computer Architecture News*, Vol. 10, No. 4, June 1982, pp. 17-27.
19. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Comm. ACM*, Vol. 27, No. 10, Oct. 1984, pp. 1013-1030.
20. R. P. Colwell, "The Performance Effects of Functional Migration and Architectural Complexity in Object-Oriented Systems," PhD. thesis, Carnegie-Mellon University, Pittsburgh, PA. Expected completion in June, 1985.
21. G. W. Cox et al., "Interprocess Communication and Processor Dispatching on the Intel 432," *ACM Trans. Computer Systems*, Vol. 1, No. 1, Feb. 1983, pp. 45-66.



Robert P. Colwell recently completed his doctoral dissertation on the performance effects of migrating functions into silicon, using the Intel 432 as a case study. His industrial experience includes design of a color graphics workstation for Perq Systems, and work on Bell Labs' microprocessors. He received the PhD and MSEE degrees from Carnegie-Mellon University in 1985 and 1978, and the BSEE degree from the University of Pittsburgh in 1977. He is a member of the IEEE and ACM.



Charles Y. Hitchcock III is a doctoral candidate in Carnegie-Mellon University's Department of Electrical and Computer Engineering. He is currently pursuing research in computer architecture and is a member of the IEEE and ACM. He graduated with honors in 1981 from Princeton University with a BSE in electrical engineering and computer science. His MSEE from CMU in 1983 followed research he did in design automation.



E. Douglas Jensen has been on the faculties of both the Computer Science and Elec-

trical and Computer Engineering Departments of Carnegie-Mellon University for six years. For the previous 14 years he performed industrial R/D on computer systems, hardware, and software. He consults and lectures extensively throughout the world and has participated widely in professional society activities.



H. M. Brinkley Sprunt is a doctoral candidate in the Department of Electrical and Computer Engineering of Carnegie-Mellon University. He received a BSEE degree in electrical engineering from Rice University in 1983. His research interests include computer architecture evaluation and design. He is a member of the IEEE and ACM.



Charles P. Kollar is a senior research staff member in Carnegie-Mellon University's computer Science Department. He is currently pursuing research in decentralized asynchronous computing systems. He has been associated with the MCF and NEBULA project at Carnegie-Mellon University since 1978. Previous research has been in the area of computer architecture validation and computer architecture description languages. He holds a BS in computer science from the University of Pittsburgh.