The Case for the Reduced Instruction Set Computer

David A. Patterson

Computer Science Division
University of California
Berkeley, California 94720

David R. Ditzel

Bell Laboratories
Computing Science Research Center
Murray Hill, New Jersey 07974

INTRODUCTION

One of the primary goals of computer architects is to design computers that are more cost-effective than their predecessors. Cost-effectiveness includes the cost of hardware to manufacture the machine, the cost of programming, and costs incurred related to the architecture in debugging both the initial hardware and subsequent programs. If we review the history of computer families we find that the most common architectural change is the trend toward ever more complex machines. Presumably this additional complexity has a positive tradeoff with regard to the cost-effectiveness of newer models. In this paper we propose that this trend is not always cost-effective, and in fact, may even do more harm than good. We shall examine the case for a Reduced Instruction Set Computer (RISC) being as cost-effective as a Complex Instruction Set Computer (CISC). This paper will argue that the next generation of VLSI computers may be more effectively implemented as RISC's than CISC's.

As examples of this increase in complexity, consider the transitions from IBM System/3 to the System/38 [Utley78] and from the DEC PDP-11 to the VAX11. The complexity is indicated quantitatively by the size of the control store; for DEC the size has grown from 256 x 56 in the PDP 11/40 to 5120 x 96 in the VAX 11/780.

REASONS FOR INCREASED COMPLEXITY

Why have computers become more complex? We can think of several reasons:

Speed of Memory vs. Speed of CPU. John Cocke says that the complexity began with the transition from the 701 to the 709 [Cocke80]. The 701 CPU was about ten times as fast as the core main memory; this made any primitives that were implemented as subroutines much slower than primitives that were instructions. Thus the floating point subroutines became part of the 709 architecture with dramatic gains. Making the 709 more complex resulted in an advance that made it more cost-effective than the 701. Since then, many "higher-level" instructions have been added to machines in an attempt to improve performance. Note that this trend began because of the imbalance in speeds; it is not clear that architects have asked themselves whether this imbalance still holds for their designs.

Microcode and LSI Technology. Microprogrammed control allows the implementation of complex architectures more cost-effectively than hardwired control [Husson70]. Advances in integrated circuit memories made in the late 60's and early 70's have caused microprogrammed control to be the more cost-effective approach in almost every case. Once the decision is made to use microprogrammed control, the cost to expand an instruction set is very small; only a few more words of control store. Since the sizes of control memories are often powers of 2, sometimes the instruction set can be made more complex at no extra hardware cost by expanding the microprogram to completely fill the control memory. Thus the advances in implementation technology resulted in cost-effective implementation of architectures that essentially moved traditional subroutines into the architecture. Examples of such instructions are string editing, integer-to-floating conversion, and mathematical operations such as polynomial evaluation.

Code Density. With early computers, memory was very expensive. It was therefore cost effective to have very compact programs. Complex instruction sets are often heralded for their "supposed" code compaction. Attempting to obtain code density by increasing the complexity of the instruction set is often a double-edged sword however, as more instructions and addressing modes require more bits to represent them. Evidence suggests that code compaction can be as easily achieved merely by cleaning up the original instruction set. While code compaction is important, the cost of 10% more memory is often far cheaper than the cost of squeezing 10% out of the CPU by architectural "innovations." Cost for a large scale cpu is in additional circuit packages needed while cost for a single chip cpu is more likely to be in slowing down performance due to larger (hence slower) control PLA's.

Marketing Strategy. Unfortunately, the primary goal of a computer company is not to design the most cost-effective computer; the primary goal of a computer company is to make the most money by selling computers. In order to sell computers manufacturers must convince customers that their design is superior to their competitor's. Complex instruction sets are certainly primary "marketing" evidence of a better computer. In order to keep their jobs, architects must keep selling new and better designs to their internal management. The number of instructions and their "power" is often used to promote an architecture, regardless of the actual use or cost-effectiveness of the complex instruction set. In some sense the manufacturers and designers cannot be blamed for this as long as buyers of computers do not question the issue of complexity vs. cost-effectiveness. For the case of silicon houses, a fancy microprocessor is often used as a draw card, as the real profit comes from luring customers into buying large amounts of memory to go with their relatively inexpensive cpu.

Upward Compatibility. Coincident with marketing strategy is the perceived need for upward compatibility. Upward compatibility means that the primary way to improve a design is to add new, and usually more complex, features. Seldom are instructions or addressing modes removed from an architecture, resulting in a gradual increase in both the number and complexity of instructions over a series of computers. New architectures tend to have a habit of including all instructions found in the machines of successful competitors, perhaps because architects and customers have no real grasp over what defines a "good" instruction set.

Support for High Level Languages. As the use of high level languages becomes increasingly popular, manufacturers have become eager to provide more powerful instructions to support them. Unfortunately there is little evidence to suggest that any of the more complicated instruction sets have actually provided such support. On the contrary, we shall argue that in many cases the complex instruction sets are more detrimental than useful. The effort to support high-level languages is laudable, but we feel that often the focus has been on the wrong issues.

Use of Multiprogramming. The rise of timesharing required that computers be able to respond to interrupts with the ability to halt an executing process and restart it at a later time. Memory management and paging additionally required that instructions could be halted before completion and later restarted. Though neither of these had a large effect on the design of instruction sets themselves, they had a direct effect on the implementation. Complex instructions and addressing modes increase the state that has to be saved on any interrupt. Saving this state often involves the use of shadow registers and a large increase in the complexity of the microcode. This complexity largely disappears on a machine without complicated instructions or addressing modes with side effects.

HOW HAVE CISC'S BEEN USED?

One of the interesting results of rising software costs is the increasing reliance on high-level languages. One consequence is that the compiler writer is replacing the assembly-language programmer in deciding which instructions the machine will execute. Compilers are often unable to utilize complex instructions, nor do they use the insidious tricks in which assembly language programmers delight. Compilers and assembly language programmers also rightfully ignore parts of the instruction set which are not useful under the given time-space tradeoffs. The result is that often only a fairly small part of the architecture is being used.

For example, measurements of a particular IBM 360 compiler found that 10 instructions accounted for 80% of all instructions executed, 16 for 90%, 21 for 95%, and 30 for 99% [Alexander75]. Another study of various compilers and assembly language programs concluded that "little flexibility would be lost if the set of instructions on the CDC-3600 were reduced to ½ or ¼ of the instructions now available."[Foster71] Shustek points out for the IBM 370 that "as has been observed many times, very few opcodes account for most of a program's execution. The COBOL program, for example, executes 84 of the available 183 instructions, but 48 represent 99.08 % of all instructions executed, and 26 represent 90.28 %."[Shustek78] Similar statistics are found when examining the use of addressing modes.

CONSEQUENCES OF CISC IMPLEMENTATIONS

Rapid changes in technology and the difficulties in implementing CISCs have resulted in several interesting effects.

Faster memory. The advances in semiconductor memory have made several changes to the assumptions about the relative difference in speed between the CPU and main memory. Semiconductor memories are both fast and relatively inexpensive. The recent use of cache memories in many systems further reduces the difference between CPU and memory speeds.

Irrational Implementations. Perhaps the most unusual aspect of the implementation of a complex architecture is that it is difficult to have "rational" implementations. By this we mean that special purpose instructions are not always faster than a sequence of simple instructions. One example was discovered by Peuto and Shustek for the IBM 370 [Peuto,Shustek77]; they found that a sequence of load instructions is faster than a load multiple instruction for fewer than 4 registers. This case covers 40% of the load multiple instructions in typical programs. Another comes from the VAX-11/780. The INDEX instruction is used to calculate the address of an array element while at the same time checking to see that the index fits in the array bounds. This is clearly an important function to accurately detect errors in high-level languages statements. We found that for the VAX 11/780, replacing this single "high level" instruction by several simple instructions (COMPARE, JUMP LESS UNSIGNED, ADD, MULTIPLY) that we could perform the same function 45%

faster! Furthermore, if the compiler took advantage of the case where the lower bound was zero, the simple instruction sequence was 60% faster. Clearly smaller code does not always imply faster code, nor do "higher-level" instructions imply faster code.

Lengthened Design Time. One of the costs that is sometimes ignored is the time to develop a new architecture. Even though the replication costs of a CISC may be low, the design time is greatly expanded. It took DEC only 6 months to design and begin delivery of the PDP-1, but it now takes at least three years to go through the same cycle for a machine like the VAX.[1] This long design-time can have a major effect on the quality of the resulting implementation; the machine is either announced with a three year old technology or the designers must try to forecast a good implementation technology and attempt to pioneer that technology while building the machine. It is clear that reduced design time would have very positive benefits on the resulting machine.

Increased Design Errors. One of the major problems of complex instruction sets is debugging the design; this usually means removing errors from the microprogram control. Although difficult to document, it is likely that these corrections were a major problem with the IBM 360 family, as almost every member of the family used read only control store. The 370 line uses alterable control store exclusively, due perhaps to decreased hardware costs, but more likely from the bad experience with errors on the 360. The control store is loaded from a floppy disk allowing microcode to be maintained similarly to operating systems; bugs are repaired and new floppies with updated versions of the microcode are released to the field. The VAX 11/780 design team realized the potential for microcode errors. Their solution was to use a Field Programmable Logic Array and 1024 words of Writable Control Store (WCS) to patch microcode errors. Fortunately DEC is more open about their experiences so we know that more than 50 patches have been made. Few believe that the last error has been found.[2]

RISC AND VLSI

The design of single chip VLSI computers makes the above problems with CISC's even more critical than with their multi-chip SSI implementations. Several factors indicate a Reduced Instruction Set Computer as a reasonable design alternative.

Implementation Feasibility. A great deal depends on being able to fit an entire CPU design on a single chip. A complex architecture has less of a chance of being realized in a given technology than does a less complicated architecture. A good example of this is DEC's VAX series of computers. Though the high end models may seem impressive, the complexity of the architecture makes its implementation on a single chip extremely difficult with current design rules, if not totally impossible. Improvement in VLSI technology will eventually make a single chip version feasible, but only after less complex but equally functional 32-bit architectures can be realized. RISC computers therefore benefit from being realizable at an earlier date.

[1] Some have offered other explanations. Everything takes longer now (software, mail, nuclear power plants), so why shouldn't computers? It was also mentioned that a young, hungry company would probably take less time than an established company. Although these observations may partially explain DEC's experiences, we believe that, regardless of the circumstances, the complexity of the architecture will affect the design cycle.

[2] Each patch means several microinstructions must be put into WCS, so the 50 patches require 252 microinstructions. Since there was a good chance of errors in the complex VAX instructions, some of these were implemented only in WCS so the patches and the existing instructions use a substantial portion of the 1024 words.

Design Time. Design difficulty is a crucial factor in the success of VLSI computer. If VLSI technology continues to at least double chip density roughly every two years, a design that takes only two years to design and debug can potentially use a much superior technology and hence be more effective than a design that takes four years to design and debug. Since the turnaround time for a new mask is generally measured in months, each batch of errors delays product delivery another quarter; common examples are the 1-2 year delays in the Z8000 and MC68000.

Speed. The ultimate test for cost-effectiveness is the speed at which an implementation executes a given algorithm. Better use of chip area and availability of newer technology through reduced debugging time contribute to the speed of the chip. A RISC potentially gains in speed merely from a simpler design. Taking out a single address mode or instruction may lead to a less complicated control structure. This in turn can lead to smaller control PLA's, smaller microcode memories, fewer gates in the critical path of the machine; all of these can lead to a faster minor cycle time. If leaving out an instruction or address mode causes the machine to speed up the minor cycle by 10%, then the addition would have to speed up the machine by more than 10% to be cost-effective. So far, we have seen little hard evidence that complicated instruction sets are cost-effective in this manner.[3]

Better use of chip area. If you have the area, why not implement the CISC? For a given chip area there are many tradeoffs for what can be realized. We feel that the area gained back by designing a RISC architecture rather than a CISC architecture can be used to make the RISC even more attractive than the CISC. For example, we feel that the entire system performance might improve more if silicon area were instead used for on-chip caches [Patterson,Séquin80], larger and faster transistors, or even pipelining. As VLSI technology improves, the RISC architecture can always stay one step ahead of the comparable CISC. When the CISC becomes realizable on a single chip, the RISC will have the silicon area to use pipelining techniques; when the CISC gets pipelining the RISC will have on chip caches, etc. The CISC also suffers by the fact that its intrinsic complexity often makes advanced techniques even harder to implement.

SUPPORTING A HIGH-LEVEL LANGUAGE COMPUTER SYSTEM

Some would argue that simplifying an architecture is a backwards step in the support of high-level languages. A recent paper [Ditzel,Patterson80] points out that a "high level" architecture is not necessarily the most important aspect in achieving a High-Level Language Computer System. A High-Level Language Computer System has been defined as having the following characteristics:

(1) *Uses high-level languages for all programming, debugging and other user/system interactions*.

(2) *Discovers and reports syntax and execution time errors in terms of the high-level language source program*.

(3) *Does not have any outward appearance of transformations from the user programming language to any internal languages*.

Thus the only important characteristic is that a combination of hardware and software assures that the programmer is always interacting with the computer in terms of a high-level language. At no time need the programmer be aware of any lower levels in the writing or the debugging of a program. As long as this requirement is met, then the goal is achieved. Thus it makes no difference in

[3] In fact, there is evidence to the contrary. Harvey Cragon, chief architect of the TI ASC, said that this machine implemented a complex mechanism to improve performance of indexed reference inside of loops. Although they succeeded in making these operations run faster, he felt it made the ASC slower in other situations. The impact was to make the ASC slower than simpler computers designed by Cray [Cragon 80].

a High-Level Language Computer System whether it is implemented with a CISC that maps one-to-one with the tokens of the language, or if the same function is provided with a very fast but simple machine.

The experience we have from compilers suggests that the burden on compiler writers is eased when the instruction set is simple and uniform. Complex instructions that supposedly support high level functions are often impossible to generate from compilers.[4] Complex instructions are increasingly prone to ~~implementing the "wrong" function as the level of the instruction increases~~. This is because the function becomes so specialized that it becomes useless for other operations.[5] Complex instructions can generally be replaced with a small number of lower level instructions, often with little or no loss in performance.[6] The time to generate a compiler for a CISC is additionally increased because bugs are more likely to occur in generating code for complex instructions.[7]

There is a fair amount of evidence that more complicated instructions designed to make compilers easier to write often do not accomplish their goal. Several reasons account for this. First, because of the plethora of instructions there are many ways to accomplish ~~a given elementary operation~~, a situation confusing for both the compiler and compiler writer. Second, many compiler writers assume that they are dealing with a rational implementation when in fact they are not. The result is that ~~the "appropriate" instruction often turns out to be the wrong choice.~~ For example, pushing a register on the stack with **PUSHL R0** is slower than pushing it with the move instruction **MOVL R0,-(SP)** on the VAX 11/780. We can think of a dozen more examples offhand, for this and almost every other complicated machine. One has to take special care not to use an instruction "because its there." ~~These problems cannot be "fixed" by different models of the same architecture~~ without ~~totally destroying either program portability or the reputation of a good compiler writer as a~~ change ~~in the relative instruction timings would require a new code generator to retain optimal code~~ generation.

The desire to support high level languages encompasses both the achievement of a HLLCS and reducing compiler complexity. We see few cases where a RISC is substantially worse off than a CISC, leading us to conclude that a properly designed RISC seems as reasonable an architecture for supporting high level languages as a CISC.

[4] Evidence for and against comes from DEC. The complex MARK instruction was added to the PDP-11 in order improve the performance of subroutine calls; because this instruction did not do exactly what the programmers wanted it is almost never used. The damaging evidence comes from the VAX; it is rumored that the VMS FORTRAN compiler apparently produces a very large fraction (.8?) of the potential VAX instructions.

[5] We would not be surprised if FORTRAN and BLISS were used as models for several occurrences of this type of instruction on the VAX. Consider the **branch if lower bit set** and **branch if lower bit clear** instructions, which precisely implement conditional branching for BLISS, but are useless for the more common branch if zero and if not zero found in many other languages; this common occurrence requires two instructions. Similar instructions and addressing modes exist which appeal to FORTRAN.

[6] Peuto and Shustek observed that the complex decimal and character instructions of the IBM and Amdahl computers generally resulted in relatively poor performance in the high end models. They suggest that simpler instructions may lead to increased performance [Peuto,Shustek77]. They also measured the dynamic occurrence of pairs of instructions; significant results here would support the CISC philosophy. Their conclusion:
"An examination of the frequent opcode pairs fails to uncover any pair which occurs frequently enough to suggest creating additional instructions to replace it."

[7] In porting the C compiler to the VAX, over half of the bugs and about a third of the complexity resulted from the complicated INDEXED MODE.

WORK ON RISC ARCHITECTURES

At Berkeley. Investigation of a RISC architecture has gone on for several months now under the supervision of D.A. Patterson and C.H. Séquin. By a judicious choice of the proper instruction set and the design of a corresponding architecture, we feel that it should be possible to have a very simple instruction set that can be very fast. This may lead to a substantial net gain in overall program execution speed. This is the concept of the Reduced Instruction Set Computer. The implementations of RISC's will almost certainly be less costly than the implementations of CISC's. If we can show that simple architectures are just as effective to the high-level language programmer as CISC's such as VAX or the IBM S/38, we can claim to have made an effective design.

At Bell Labs. A project to design computers based upon measurements of the C programming language has been under investigation by a small number of individuals at Bell Laboratories Computing Science Research Center for a number of years. A prototype 16-bit machine was designed and constructed by A.G. Fraser. 32-bit architectures have been investigated by S.R. Bourne, D.R. Ditzel, and S.C. Johnson. Johnson used an iterative technique of proposing a machine, writing a compiler, measuring the results to propose a better machine, and then repeating the cycle over a dozen times. Though the initial intent was not specifically to come up with a simple design, the result was a RISC-like 32-bit architecture whose code density was as compact as the PDP-11 and VAX [Johnson79].

At IBM. Undoubtedly the best example RISC is the 801 minicomputer, developed by IBM Research in Yorktown Heights, N.Y.[Electronics76] [Datamation79]. This project is several years old and has had a large design team exploring the use of a RISC architecture in combination with very advanced compiler technology. Though many details are lacking their early results seem quite extraordinary. They are able to benchmark programs in a subset of PL/I that runs about five times the performance of an IBM S/370 model 168. We are certainly looking forward to more detailed information.

CONCLUSION

There are undoubtedly many examples where particular "unique" instructions can greatly improve the speed of a program. Rarely have we seen examples where the same benefits apply to the system as a whole. For a wide variety of computing environments we feel that careful pruning of an instruction set leads to a cost-effective implementation. Computer architects ought to ask themselves the following questions when designing a new instruction set. If this instruction occurs infrequently, is it justifiable on the grounds that it is necessary and unsynthesizable, for example, a Supervisor Call instruction. If the instruction occurs infrequently and is synthesizable, can it be justified on the grounds that it is a heavily time consuming operation, for example, floating point operations. If the instruction is synthesizable from a small number of more basic instructions, what is the overall impact on program size and speed if the instruction is left out? Is the instruction obtainable for free, for example, by utilizing unused control store or by using an operation already provided by the ALU? If it is obtainable for "free", what will be the cost in debugging, documentation, and the cost in future implementations? Is it likely that a compiler will be able to generate the instruction easily?

We have assumed that it is worthwhile to minimize the "complexity" (perhaps measured in design time and gates) and maximize "performance" (perhaps using average execution time expressed in gate delays as a technology-independent time unit) while meeting the definition of a High-Level Language Computer System. In particular, we feel that VLSI computers will benefit the most from the RISC concepts. Too often, the rapid advancements in VLSI technology have been used as a panacea to promote architectural complexity. We see each transistor as being precious for at least the next ten years. While the trend towards architectural complexity may be one path towards improved computers, this paper proposes another path, the Reduced Instruction Set Computer.

ACKNOWLEDGEMENTS

For their prompt and constructive comments on this paper, the authors wish to express thanks to A.V. Aho, D. Bhandarkar, R. Campbell, G. Corcoran, G. Chesson, R. Cmelik, A.G. Fraser, S.L. Graham, S.C. Johnson, P. Kessler, T. London, J. Ousterhout, D. Poplawski, M. Powell, J. Reiser, L. Rowe, B. Rowland, J. Swensen, A.S. Tanenbaum, C. Séquin, Y. Tamir, G. Taylor, and J. Wakerly. Those students at Berkeley who have participated in the RISC project are acknowledged for their excellent work. RISC research at Berkeley was sponsored in part by the Defense Advance Research Projects Agency (DoD), ARPA Order No. 3803, and monitored by Naval Electronic System Command under Contract No. N00039-78-G-0013-0004.

REFERENCES

[Alexander75] W.C. Alexander and D.B. Wortman, "Static and Dynamic characteristics of XPL Programs," *Computer,* pp. 41-46, November 1975, Vol. 8, No. 11.

[Cocke80] J. Cocke, *private communication,* February, 1980.

[Cragon80] H.A. Cragon, in his talk presenting the paper "The Case Against High-Level Language Computers," at the International Workshop on High-Level Language Computer Architecture, May 1980.

[Datamation79] *Datamation,* "IBM Mini a Radical Departure," October 1979, pp. 53-55.

[Ditzel,Patterson80] "Retrospective on High-Level Language Computer Architecture," *Seventh Annual International Symposium on Computer Architecture,* May 6-8, 1980, La Baule, France.

[Electronics76] *Electronics Magazine,* "Altering Computer Architecture is Way to Raise Throughput, Suggests IBM Researchers," December 23, 1976, pp. 30-31.

[Foster71] C.C. Foster, R.H. Gonter and E.M. Riseman, "Measures of Op-Code Utilization," *IEEE Transactions on Computers,* May, 1971, pp. 582-584.

[Husson70] S.S. Husson, *Microprogramming: Principles and Practices,* Prentice-Hall, Engelwood, N.J., pp. 109-112, 1970.

[Johnson79] S.C. Johnson, "A 32-bit Processor Design," Computer Science Technical Report # 80, Bell Labs, Murray Hill, New Jersey, April 2, 1979.

[Patterson,Séquin80] D.A. Patterson and C.H. Séquin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Journal of Solid-State Circuits, IEEE Transactions on Computers,* Joint Special Issue on Microprocessors and Microcomputers, Vol. C-29, no. 2, pp. 108-116, February 1980.

[Peuto,Shustek77] B.L. Peuto and L.J. Shustek, "An Instruction Timing Model of CPU Performance," *Conference Proc., Fourth Annual Symposium on Computer Architecture,* March 1977.

[Shustek78] L.J. Shustek, "Analysis and Performance of Computer Instruction Sets," Stanford Linear Accelerator Center Report 205, Stanford University, May, 1978, pp. 56.

[Utley78] B.G. Utley *et al,* "IBM System/38 Technical Developments," IBM GS80-0237, 1978.

ERRATA

The footnote at the bottom of page 29 in "The Case for the Reduced Instruction Set Computer" contains an error. Harvey Cragon was speaking on reducing the semantic gap by making computers more complicated. The most appropriate slide is quoted directly below:

> "... When doing vector operations the performance was approximately the same on these two machines (the CDC 7600 and the TI ASC). Memory bandwidth was approximately equal. The buffering of the hardware DO LOOP accomplished the same memory bandwidth reduction as did the buffering of the normal instruction stream on the 7600. After the proper macros had been written for the 7600, and the calling procedure incorporated into the compiler, equal access to the vector capability was provided from FORTRAN. Due in large measure to the complexity introduced by the vector hardware, scalar performance on the ASC is less than the 7600. The last, and most telling argument, is that more hardware was needed on the ASC than was required on the 7600. The additional hardware for building the rather elaborate DO LOOP operations did not have the pay-off that we had anticipated.
>
> The experience with the ASC, and other experiences, have led me to question all the glowing promises which are made if we will only close the semantic gap. The promised benefits do not materialize, generality is lost."

Clearly the relative speeds of the two depend upon the mix of scalar and vector operations in a given application.

We also inadvertently changed Harvey's middle initial: he is Harvey G. Cragon.

Dave Patterson
Dave Ditzel