A RISCy APPROACH TO COMPUTER DESIGN

by David A. Patterson

Computer Science Division
University of California
Berkeley, California 94720

Introduction

Computer architects are facing two major questions:

What do you do with all those transistors?

How do you support High-level languages (HLL)?

Most architects argue that support of HLL implies the addition of instructions that look like HLL statements. They also argue that you need a very rich instruction set to allow a variety of cost-performance implementations. Examples of this philosophy towards increasing architectural complexity are the DEC VAX 11 family[1], the military standard computer family Nebula[2], and the Intel iAPX-432. [3]

Complexity has its cost in a longer design cycle and in less effective use of VLSI technology. [4] The cost of delay of one successful computer has been estimated as one million dollars per day. [5] Hanover argues a successful computer product has essentially an unlimited number of customers. The only limit is the product lifetime i.e., from introduction until it becomes technically obsolete. Since the VAX 11/780 produces $1M per day in sales, delaying introduction to market would have cost DEC $1M per day.

In addition to slowing the introduction of new computers, the added architectural complexity may not benefit HLL programs. The compiler writer must balance the potential performance gains of new instructions with the added compiler complexity and increased compile time needed to use the new instructions. Thus it is not unusual for a HLL compiler to generate less than half of the instructions of an architecture. [6]

Our conclusion is that architects are proposing more complex machines that are more expensive and difficult to build, and these complex machines can only be programmed effectively in assembly language. We call this class of computer Complex Instruction Set Computers (CISC). Drawing an analogy from the automotive industry, Figure 1 is our symbol of CISC's.

An alternative to this design style suggests that simplicity is a better match to VLSI and HLL. Simplifying the design reduces the elapsed design time and thus makes it considerably easier to track the VLSI growth curve. Simplifying the instruction set avoids dedicating hardware to features that cannot be used by HLL compilers; this hardware savings can be dedicated to increasing the performance of the simpler instruction set. We call this alternative class Reduced Instruction Set Computers (RISC). [4] Examples are the 801 [7,8] at IBM, RISC I at Berkeley, and MIPS [9] at Stanford. Figure 2 is our automotive symbol of a RISC.

This paper describes RISC I, a single VLSI chip RISC developed at Berkeley. An architectural summary is followed by a description of our scheme to support HLL procedure calls. The VLSI design of RISC I is compared to other microprocessors and the performance of RISC I is compared to microprocessors and minicomputers. We conclude with a discussion of some of the controversial issues associated with RISC's. Detailed information about RISC I has been published elsewhere[10,11,12] and there are several other articles discussing RISC's versus CISC's. [4,13,14,15,16]

Architecture

At the onset of the design of RISC I we defined the following goals and constraints: (a) find a reasonable compromise between high performance for high-level language programs and a simple, single chip implementation; (b) make the size of all instructions equal to one word and execute all instructions in one machine cycle; (c) emphasize register oriented instructions and restrict memory access to the LOAD and STORE instructions. The resulting architecture has 31 instructions in two formats, uses 32-bit addresses, and supports 8-,16-, and 32-bit data. Figure 3 shows the data paths of RISC I.

The simple instruction set gave us two choices: make a less expensive machine by implementing RISC I with less hardware or make a higher performance RISC. We chose the latter course. In particular, we decided to improve the performance of the HLL procedure call, which is probably the most expensive HLL construct. [10,17,18]

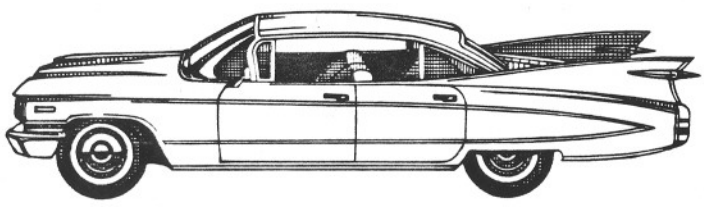

Figure 1. CISC Symbol

nust not extend below this line

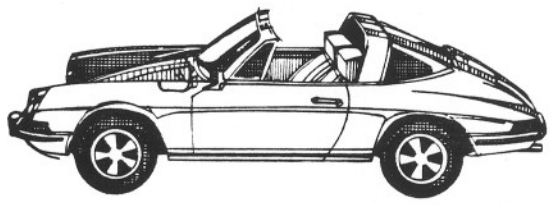

Figure 2. RISC Symbol

TO APPEAR AT COMPCON, FEBRUARY 1982
SAN FRANCISCO

HLL's generally perform the following operations in procedure call:

(1) Push parameters onto a stack in memory;
(2) Save the return address in memory;
(3) Jump to procedure;
(4) Save registers to be restored after the call;
(5) Allocate a set of local variables on the stack.

Return from procedure simply reverses these operations.

An obvious first step in improving performance is to remove the need for (4). One solution would be to remove registers from the architecture. We decided instead to have several sets, or windows, of registers. The programmer gets a new window of registers on procedure call and gets the old window of registers on procedure return. If it is convenient to have some registers that are not affected by procedure call, then the registers can be divided into two groups: *Local* (a new set every call) and *Global* (not effected by calls).

It is not feasible to limit the the depth of nested calls, so we use these sets of registers to buffer the saving and restoring of register windows. We dump the buffer into memory when it is full and load it from memory when it is empty. This scheme would not work if programs normally have long sequences of calls followed by long sequences of returns. Halbert and Kessler found that there is "locality" of procedure nesting as shown in Table 1. [19] Based on nine large recursive C programs, it appears that 6 to 8 windows is a sufficient buffer size. We chose 8 in RISC I.

This improvement still requires memory accesses in steps (1) and (2) of call plus it requires memory accesses for every parameter or local variable. Halbert and Kessler proposed having a larger set of registers with registers dedicated to the different types of storage needed by a procedure. RISC I splits the 32 visible registers into 4 groups:

Local(R16-R25): Used for variables local to a procedure;

High(R26-R31): Used to hold the parameters being passed to the current procedure;

Low (R10-R15): Used by the current procedure to pass parameters to a procedure that is being called;

Global(R0-R9): Used for global variables and not affected by procedure calls.

Note that the low registers of the calling procedure must become the high registers of the called procedure. This can be accomplished by simply adjusting the pointer to a window of registers so that the low registers of the calling frame overlap the high registers of the calling frame. Figure 4 illustrates the approach. In this figure procedure A calls procedure B and B calls C. A parameter shared between A and B is considered as a low register of A and a high register of B, e.g., R15 in A is the same as R31 in B. We call this scheme *overlapped register windows*.

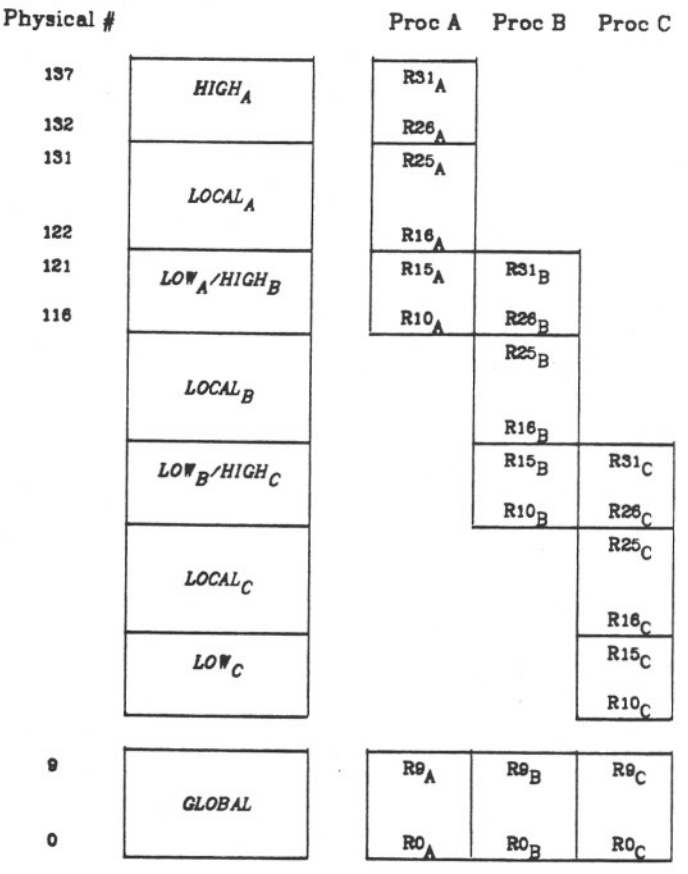

Figure 4. Three Overlapped Register Windows

The procedure call in RISC now becomes:

(1) Put the parameters into the LOW registers;
(2) Save the return address in a LOW register;
(3) Jump to the procedure.

If there are too many parameters or locals, the RISC I compiler simply allocates the rest in a stack in memory. This does not occur frequently as 6 parameters and 10 locals will handle 98% of the procedure calls in the C programs listed in Table 1. [19]

Overlapped register windows gives RISC a typical call time of 2 μsec versus 20 μsec for a typical call on the VAX 11/780. In addition, overlapped register windows improve performance of all instructions since local and parameter variables are found in registers instead of memory. This scheme reduces overall accesses to data memory by a factor of two. [10]

TABLE 1. Percent of procedure calls that required register frame saves.

Program	Calls	Number of windows								
		2	4	6	8	10	12	16	24	32
cc - C compiler	96610	46.3	14.4	6.4	3.0	1.6	0.9	0.2	0.0+	0.0+
cp - copy file	54	3.7	0.0	-	-	-	-	-	-	-
finger - find person	7353	11.8	1.3	0.0+	-	-	-	-	-	-
more - print text	1137	16.8	1.7	0.1	-	-	-	-	-	-
pi - pascal interp.	37865	43.5	10.4	1.8	0.9	0.6	0.4	0.2	0.0+	-
printenv	692	3.5	1.3	0.1	-	-	-	-	-	-
sort	3659	5.6	0.3	0.1	0.0+	-	-	-	-	-
troff - typeset text	159542	44.5	8.6	2.8	1.3	0.4	0.2	0.0+	-	-
w - who's logged in	2948	19.5	1.2	0.0+	-	-	-	-	-	-

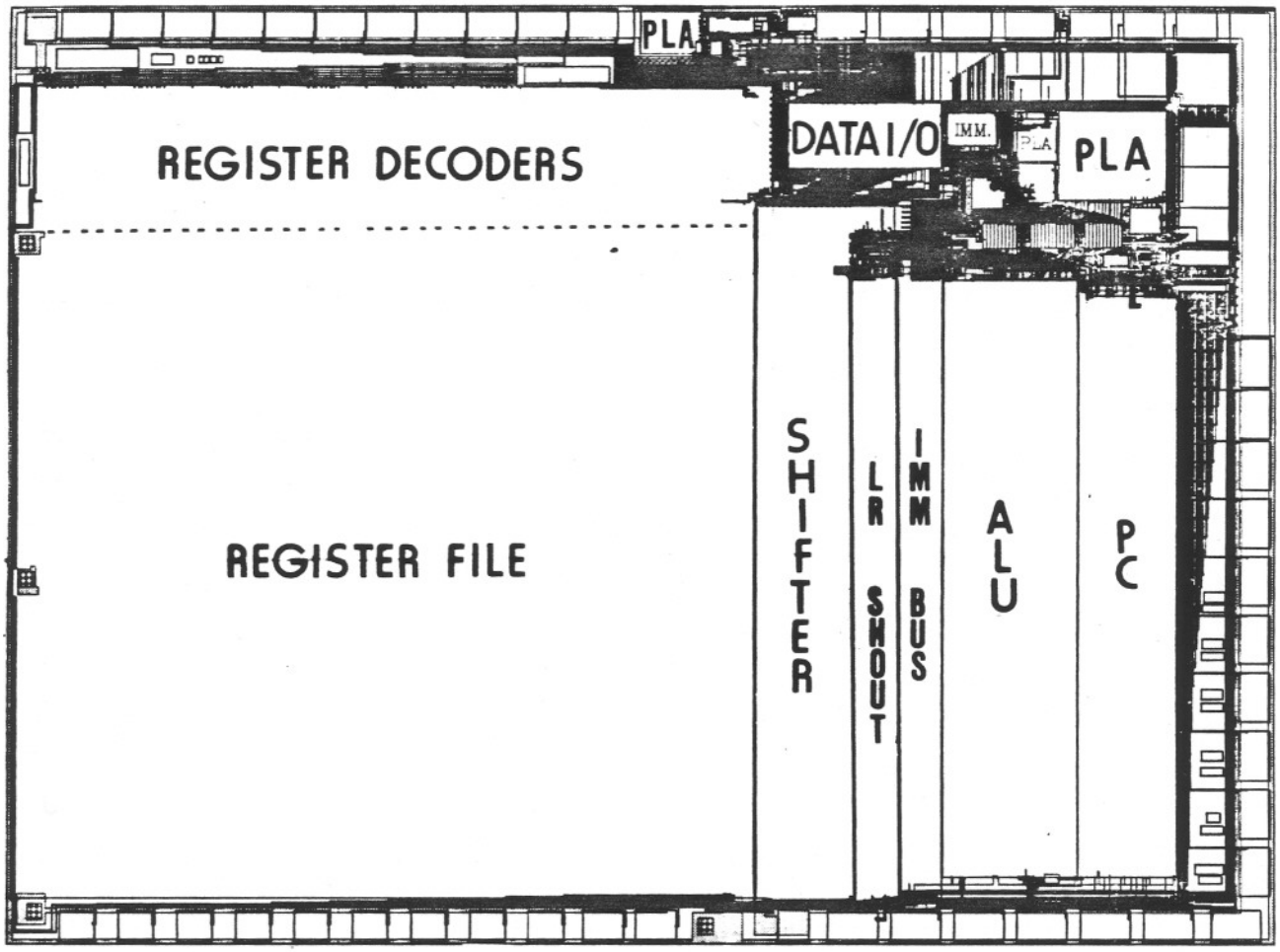

Figure 5. Chip Plan of RISC I

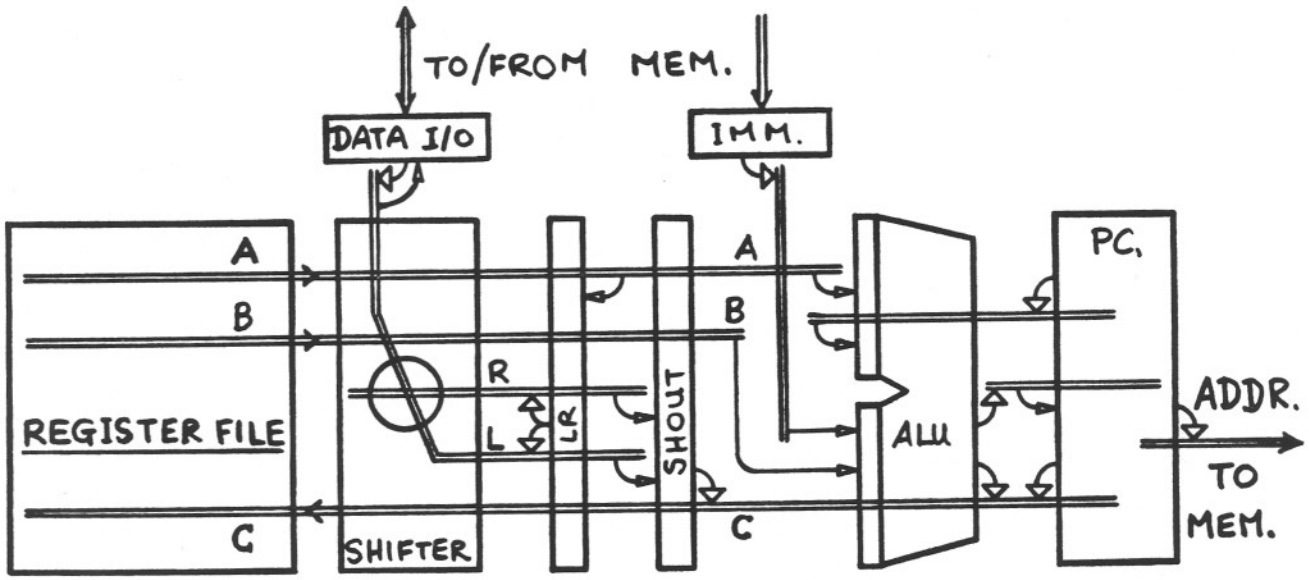

Figure 3. RISC I Data Paths

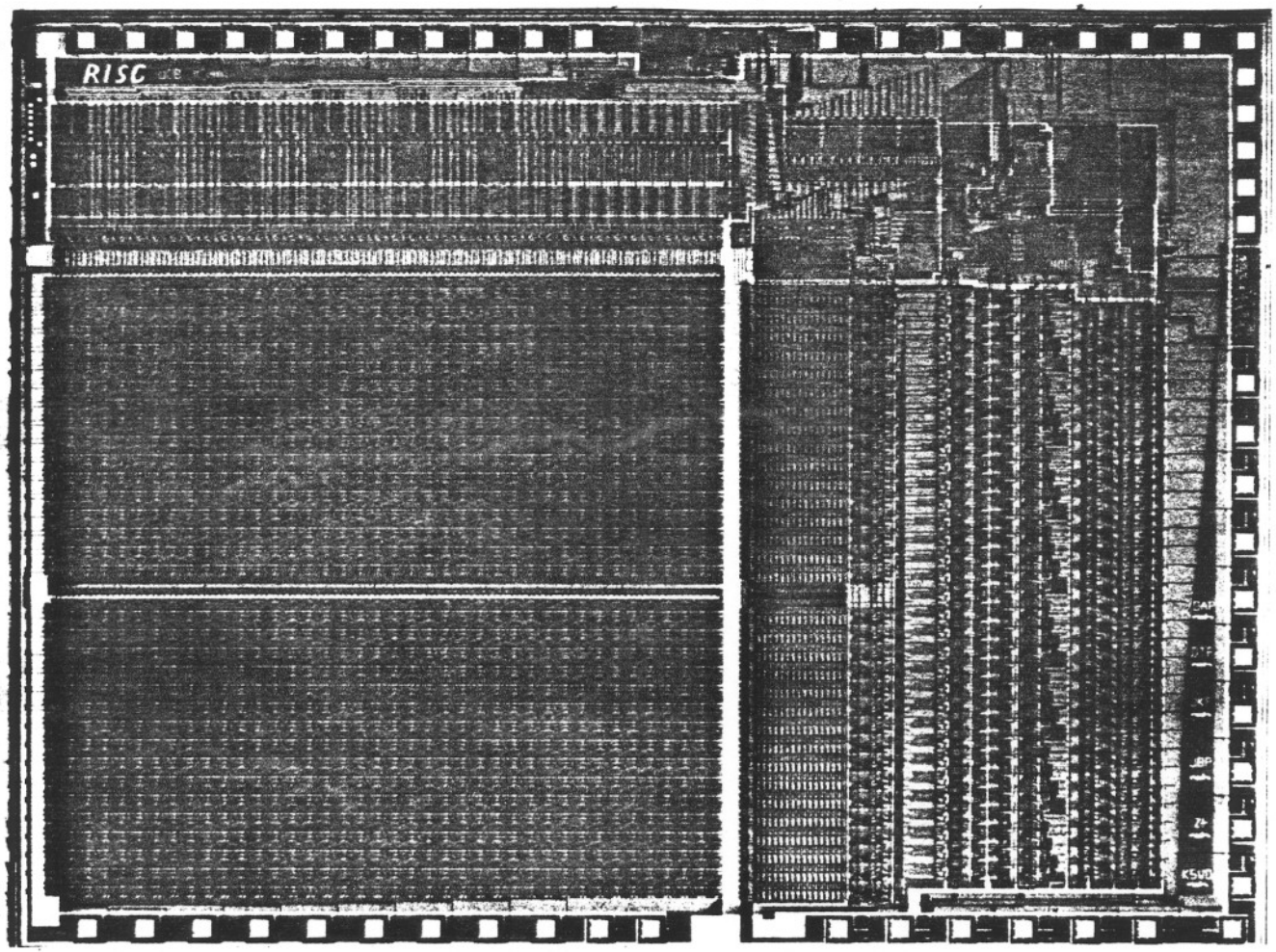

Figure 6. Photomicrgraph of RISC I

TABLE 2. Design metrics for Z8000, MC68000, iAPX-432, and RISC I.

	Zilog Z8000	Motorola 68000	Intel iAPX-432 43201	43202	43203	RISC I
Total Devices	17.5k	68k	110k	49k	60k	44k
Total minus ROM	17.5k	37k	44k	49k	44k	44k
Drawn Devices	3.5k	3.0k	5.6k	9.5k	5.7k	1.8k
Regularization factor	5.0	12.1	7.9	5.2	7.7	25
Size of chip (mils)	238x251	246x281	318x323	366x313	358x326	406x305
(Area in mil^2)	60k	69k	103k	115k	117k	124k
Size of Control (mil^2)	37k	42k	67k	45k	47k	7k
Percent Control	53 %	62 %	65 %	39 %	40 %	6 %
Elapsed Time to first silicon (months)	30	30	33	33	21	19
Design Time (man months)	60	100	170	170	130	15
Layout Time (man months)	70	70	90	100	50	12

ALL MATERIAL IN THIS SPACE WILL BE DELETED

VLSI Implementation

Circuit design of RISC I began January 6, 1981 and mask descriptions were completed June 22, 1981. The chips were fabricated over the summer and we received first silicon October 23, 1981. We followed the Mead-Conway design philosophy for NMOS with lambda at 2 microns and no buried contacts. Figure 5 shows the area occupied by the blocks in Figure 3 and Figure 6 is a photomicrograph of RISC I.

We collected statistics on the design and layout of RISC I. [12] Table 2 compares these results to the more complex architectures implemented in VLSI. The most visible impact of the reduced instruction set is the reduced control area: control (labeled 'PLA' in Figure 4) is only 6 % of RISC I compared to 50 % in others. RISC I also more regular. Lattin defined the *regularity factor* as the total number of transistors (less those in ROM) divided by the number of individually drawn transistors. [20] By this measure RISC I is 2 to 5 times more regular than the Z8000, 68000, or 432. The time from the first discussion of the RISC I architecture to first silicon was 19 months, only half the time of other machines. The reduced design and layout effort was due in part to the reduced instruction set and in part to the good Berkeley CAD software. The primary interface was Caesar, an excellent color graphics mask editor developed by Ousterhout. [21]

As we go to press, we are just testing the RISC I chips. We know that there was improper processing of the polysilicon layer. We are trying to determine whether the behavior of the chip is due to processing mistakes or due to our design errors. We now have a better layout rules checker and have found a about a half-dozen layout rules violations, but none explain the varying behavior of the chips. We have not found any circuit design errors. The only repeatable phenomenon is that power consumption is consistently less than 700 milliwatts, about half our original estimate.

Performance

Prototype versions of a RISC I C compiler, optimizer, linker, assembler, and simulator have been developed to predict the performance of RISC I on HLL programs. Our first result is that RISC programs are typically 50% larger than the corresponding VAX programs. [10,12] This is better than we had expected since we consciously ignored optimization of program size.

Our main architectural goal was high performance. Table 3 compares the the relative performance of several minicomputers and microprocessors on eleven C programs. The first five programs are HLL versions of the "EDN" benchamrks. [22] The other C programs range from toy programs (e.g., towers of hanoi) to programs from the UNIX environment that are used every day (e.g., sed, a batch-oriented text editor). The minicomputers under test are the VAX-11/780, a 32-bit Schottky-TTL machine with a 200 ns microcycle time; the PDP 11/70, a 16-bit Schottky-TTL machine with a 135 ns cycle time; and the BBN C/70, a 20-bit Schottky-TTL machine with a 150 ns microcycle time. Both the VAX and PDP-11 have a cache. The microprocessors under test are RISC I, Z8001, and MC68000. RISC I is a 32-bit microprocessor with 32-bit addresses that uses NMOS technology nearly as good as as that used by the Z8001 and 68000. The Z8001 is a 16-bit computer with a 16-bit address while the 68000 works effectively as either a 16-bit or 32-bit computer with a 24-bit address. Both the MC68000 and Z8001 come with a variety of clock rates; we assumed 10 MHz for the 68000 and 6 MHz for the Z8001. RISC I is designed to run at 400 ns per instruction (read 2 registers, 32-bit add, write register, and prefetch the next instruction) which implies the same speed memory as the 10 MHz 68000. We don't have working hardware for the either the 68000 or RISC I, so we used simulators to predict performance. Based solely on technology, one would expect the performance to follow the order VAX 11/780, PDP 11/70, C/70, MC68000, Z8001 with RISC I being in the same performance range as the MC68000 and Z8001. As we can see from Table 3, RISC I does not follow expectations.

TABLE 3. C Benchmarks: RISC I Execution Time and RISC I Performance Ratio (± standard deviation).

BENCHMARK	RISC I	VAX 11/780	11/70	68000	Z8001	C/70
	msecs	Number of Times Slower Than RISC I				
E - string search	.46	1.3	0.9	2.8	1.6	2.2
F - bit test	.06	4.8	6.2	4.5	7.2	9.2
H - linked list	.10	1.2	1.9	1.6	2.4	2.5
K - bit matrix	.43	3.0	4.0	4.0	5.2	9.3
I - quicksort	50.4	3.0	3.6	4.1	5.2	5.8
Ackermann(3,6)	3200	1.6	1.6	---	2.8	---
recursive qsort	800	2.3	3.2	---	5.9	1.3
puzzle(subscript)	4700	2.0	1.6	---	4.2	3.4
puzzle(pointer)	3200	1.3	2.0	4.2	2.3	2.1
sed(batch editor)	5100	1.1	1.1	---	4.4	2.6
towers hanoi(18)	6800	1.8	2.3	---	4.2	1.6
Average		2.1 ± 1.1	2.6 ± 1.5	3.5 ± 1.8	4.1 ± 1.6	4.0 ± 2.8

Discussion

It is not surprising that an idea as unconventional as RISC results in controversy. Listed below are frequently heard comments *(in italics)* followed by a short discussion of that comment.

> *CISC's have raised the "level" of the architecture by including HLL primitives (CASE,CALL) while RISC's have lowered the level. Thus CISC's provide better support of HLL.*

The CISC approach to architectural support for HLL is to narrow the gap between semantics of the assembly language and the semantics of a HLL. However, support can also be viewed as how expensive is it to use a HLL on an architecture. If the architect provides a feature that "looks" like the HLL construct, but it runs very slowly, it is likely that (a) the compiler writer will not use that architectural feature, or, if the compiler writer does use the feature, then (b) the HLL programmer will realize that a HLL feature is very slow and will avoid the feature in his HLL programs. A recent study indicates that CISC's do far more to penalize the use of HLL than RISC's. [23]

> *Since a compiler must produce more RISC instructions than CISC instructions for each HLL statement, it must be more difficult to build a RISC compiler than a CISC compiler.*

A recent paper by Wulf [24] helps explain why this is not true. He says that compiling is essentially a large "case analysis." The more *ways* there are to do something (more instructions), the more cases must be considered. Since the compiler writer must balance the speed of the compiler with his desire to get good code, he may not have the time to perform the case analysis necessary to generate all of the CISC instructions. This explains Wulf's recommendation that architectures provide either one way or every way to perform an operation. In RISC there is generally only one way to perform a given operation, e.g., if an operand is in memory it must be loaded into a register. Simple case analysis implies a simple compiler even if more instructions must be generated in each case. If a CISC provides several different ways to do an operation and thus more cases, the compiler will be more complicated, even if only a few instructions are generated for each case.

> *RISC I is tailored to C and will not perform well with other HLL.*

The first response is that other RISC's have demonstrated the viability of Pascal (MIPS) and a PL/I-like language (801). Perhaps a better reply is that studies of HLL's [6,25] indicate that the most frequently executed operations are the simple HLL operations that RISC's perform very effectively. Unless a HLL significantly changes the way people program, we expect to see similar results.

> *The comparisons of the six computers are somewhat unfair in that only the VAX provides a virtual address space that is larger than the physical address space. RISC would be much slower if it had virtual memory.*

To answer the question "How much slower?" we looked at solutions used by other microprocessors. National Semiconductor has announced the 16082, a memory management chip with an address cache that normally translates virtual addresses into physical addresses in 100 ns. They claim that only 3% of the memory addresses will need extra memory accesses because of misses in the address cache. If we were to put this chip in a system with a RISC CPU it would add another 100 ns to every memory access. Since memory is referenced every 400 ns in RISC I, such a combination would reduce RISC performance by 25%. A more sophisticated approach would be to reduce or eliminate the virtual memory overhead for instruction accesses. Since 80% to 90% of the memory references in RISC I are to instructions, [10] techniques such as translating instruction addresses only when a page boundary is crossed (used in the VAX 11/780) or providing a virtual address cache (used in the Dorado) may be very effective. This technique would reduce the overhead for virtual address translation to about 5%. A final observation is that even if the addition of virtual memory cut the performance of NMOS RISC I in half, it would still have performance comparable to the VAX implemented in Shottky TTL. It does not seem likely that virtual memory considerations will significantly change the results.

> *The primary reason for the small percent of control is the large register set.*

Comparison of the absolute sizes of control in Table 2 shows that RISC I is smaller by a factor of 5 to 10. Furthermore, even if RISC had no registers at all, control would be only 12 % of the area.

> *The reason for the good performance is the overlapped register windows. The reduced instruction set has nothing to do with it.*

Certainly a significant portion of the speed is due to the overlapped register windows of RISC I, but that does not explain the performance of the 801 or MIPS. Also, keep in mind that there would have been no room for register windows if control had not dropped from 50% to 6%.

> *These results just apply to a particular technology and architecture: you cannot generalize a RISC into a family of computers with different cost performance.*

It is true that both RISC and MIPS are NMOS designs, but the 801 was built in ECL and competes with very large computers. [7,16] We have done paper studies of RISC I in other technologies and the advantages of less hardware with higher performance still apply. We will have to wait for others to build RISC's in other technologies before we can tell if this comment is valid.

Conclusion

RISC I is a representative of a new style of computers that take less time to build and yet provide high performance. While traditional machines "support" HLL with instructions that look like HLL constructs, this machine supports the use of HLL with instructions that HLL compilers can use efficiently.

This research area is by no means closed. Some of the topics to be investigated include the applicability of RISC to other HLL's (e.g., LISP, COBOL, ADA), the performance of an operating system on RISC (e.g., UNIX), the architecture of co-processors for RISC (e.g., graphics, floating point), migration of software to RISC (e.g., a 370 emulator written in RISC machine language), and the implementation of RISC in other technologies (CMOS, TTL, ECL). This list is too big for one university; we are very interested in helping industry and academia explore RISCy architectures.

Acknowledgement

Prof. Carlo Séquin shares the management of the RISC project and shares the authorship of most of the RISC papers, but is solely to blame for the RISC acronym. Prof. John Ousterhout created, maintained, and revised Caesar, our principle design aid, and consistently provided useful technical and editorial advice. Lloyd Dickman was actively involved with the design of RISC during his sabbatical at Berkeley and supplied technical and managerial expertise. I also want to thank Prof. Richard Newton for dedicating his VLSI class to the RISC project.

The RISC research was investigated over a four quarter sequence of graduate courses at Berkeley. I would like to thank every student who participated, and give special thanks to a few. Manolis Katevenis did the initial block structure, the initial timing description and provided many important simplifications and ideas about the implementation and the architecture. Jim Peek, Korbin Van Dyke, John Foderaro, Dan Fitzpatrick, and Zvi Peshkess were the principal VLSI designers of the RISC chip. Ralph Campbell wrote the initial C compiler, the optimizer, assembler, and linker. Michael Arnold, Dan Fitzpatrick, John Foderaro, and Howard Landman all wrote CAD tools that were crucial to the VLSI implementation of RISC I. Yuval Tamir wrote a simulator and provided many suggestions in the initial design of RISC I. Peter Kessler helped derive the overlapped register windows and helped with the CAD software. Bob Sherburne is currently working with Katevenis on a more efficient VLSI implementation of RISC.

I would also like to thank John Ousterhout, Carlo Séquin, and Manolis Katevenis for their useful suggestions in improving this paper.

This research was supported in part by Bell Laboratories and in part by 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. We would like to thank Duane Adams, Paul Losleben, and DARPA for providing the resources that allow universities to attempt projects involving high-risk.

References

1. Strecker, W.D., "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proc. NCC*, pp. 967-980 (June 1978).
2. Szewerenko, L., Dietz, W.B., and Ward, P.E., "Nebula: A New Architecture and Its Relationship to Computer Hardware," *Computer* **14**(2) pp. 35-41 (February 1981).
3. Lattin, W.W., Bayliss, J.A., Budde, D.L., Colley, S.R., Cox, G.W., Goodman, A.L., Rattner, J.R., Richardson, W.S., and Swanson, R.C., "A 32b VLSI Micromainframe Computer System," *Proc. IEEE International Solid-State Circuits Conference*, pp. 110-111 (February, 1981).
4. Patterson, D.A. and Ditzel, D.R., "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* **8**(6) pp. 25-33 (15 October 1980).
5. Hanover, A., *Advanced Design Aid Development at DEC*, U.C. Berkeley Public Lecture. November 20, 1981.
6. Alexander, W.C. and Wortman, D.B., "Static and Dynamic characteristics of XPL Programs," *Computer* **8** (11) pp. 41-46 (November 1975).
7. Datamation,, "IBM Mini a Radical Departure," *Datamation*, pp. 53-55 (October 1979).
8. Radin, G., "The 801 Minicomputer," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems* , (March 1-3, 1982).
9. Hennessy, J., Jouppi, N., Baskett, F., Strong, A., Gross, T., Rowen, C., and Gill, J., "The MIPS Machine," *Compcon*, (February 1982). In this proceedings.
10. Patterson, D.A. and Séquin, C.H., "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. Eighth International Symposium on Computer Architecture*, pp. 443-457 (May 1981).
11. Séquin, C.H. and Patterson, D.A., "The Reduced Instruction Set Computer," *International Seminar on the Teaching of Computing Science*, (September 8-11, 1981).
12. Fitzpatrick, D.T., Foderaro, J.K., Katevenis, M.G.H., Landman, H.A., Patterson, D.A., Peek, J.B., Peshkess, Z., Séquin, C.H., Sherburne, R.W., and Van Dyke, K.S., "A RISCy Approach to VLSI," *VLSI Design*, pp. 14-20 (Fourth Quarter (October, 1981)).
13. Clark, D.W. and Strecker, W.D., "Comments on 'The Case for the Reduced Instruction Set Computer'," *Computer Architecture News* **8**(6) pp. 34-38 (15 October 1980).
14. Denning, P., "Computer Architecture: Some Old Ideas that Haven't Quite Made It Yet," *Comm. ACM* **24**(9) pp. 553-554 (September 1981).
15. Taylor, A., "Can Simpler Computers Also Be Better?," *Computer World* **XV**(27) p. 33 (July 6, 1981).
16. Bernhard, R., "More Hardware Means Less Software," *IEEE Spectrum* **18**(12) pp. 30-37 (December 1981).
17. Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architecture," *Comm. ACM* **20**(3) pp. 143-153 (March 1977).
18. Wichmann, B.A., "Ackermann's Function: A Study in the Efficiency of Calling Procedures," *BIT* **16**(1) pp. 103-110 (1976).
19. Halbert, D. and Kessler, P., *Windows of Overlapping Registers*, CS292R Final Reports June 9, 1980.
20. Lattin, W.W., Bayliss, J.A., Budde, D.L., Rattner, J.R., and Richardson, W.S., "A Methodology for VLSI Chip Design," *Lambda - The Magazine of VLSI Design*, pp. 34-44 (Second Quarter, 1981).
21. Ousterhout, J.K., "Caesar: An Interactive Editor for VLSI Layout," *Compcon*, (February 1982). In this proceedings.
22. Grappel, R.G. and Hemmengway, J.E., "A Tale of Four Microprocessors: Benchmarks Quantify Performance," *Electronic Design News*, pp. 179-265 (April 1, 1981).
23. Patterson, D.A. and Piepho, R.S., "RISC Assessment: A High-Level Language Experiment," *Proc. Ninth International Symposium on Computer Architecture*, (April 26-29, 1982). Submitted for publication.
24. Wulf, W.A., "Compilers and Computer Architecture," *Computer* **14**(7) pp. 41-48 (July 1981).
25. Ditzel, D.R., "Program Measurements on a High-Level Language Computer," *Computer* **13**(8) pp. 62-72 (August 1980).