RISC I: A REDUCED INSTRUCTION SET VLSI COMPUTER

DAVID A. PATTERSON and CARLO H. SEQUIN

Computer Science Division
University of California
Berkeley, California

ABSTRACT

The Reduced Instruction Set Computer (RISC) Project investigates an alternative to the general trend toward computers with increasingly complex instruction sets: With a proper set of instructions and a corresponding architectural design, a machine with a high effective throughput can be achieved. The simplicity of the instruction set and addressing modes allows most instructions to execute in a single machine cycle, and the simplicity of each instruction guarantees a short cycle time. In addition, such a machine should have a much shorter design time.

This paper presents the architecture of RISC I and its novel hardware support scheme for procedure call/return. Overlapping sets of register banks that can pass parameters directly to subroutines are largely responsible for the excellent performance of RISC I. Static and dynamic comparisons between this new architecture and more traditional machines are given. Although instructions are simpler, the average length of programs was found not to exceed programs for DEC VAX 11 by more than a factor of 2. Preliminary benchmarks demonstrate the performance advantages of RISC. It appears possible to build a single chip computer faster than VAX 11/780.

INTRODUCTION

A general trend in computers today is to increase the complexity of architectures commensurate with the increasing potential of implementation technologies, as exemplified by the complex successors of simpler machines. Compare, for example, VAX 11[1] to PDP-11, IBM System/38[2] to IBM System/3, and Intel iAPX-432[3] to 8086. The consequences of this complexity are increased design time, increased design errors, and inconsistent implementations.[4] We call this class of computers, complex instruction set computers (CISC).

Investigations of VLSI architectures[5] indicated that one of the major design limitations is the delay-power penalty of data transfers across chip boundaries and the still-limited amount of resources (devices) available on a single chip. Even a million transistors does not go far if a whole computer has to be built from it.[6] This raises the question as to whether the extra hardware needed to implement CISC is the best way to use this "scarce" resource.

The above findings led to the Reduced Instruction Set Computer (RISC) Project. The purpose of the project is to explore alternatives to the general trend toward architectural complexity. The hypothesis is that by reducing the instruction set, VLSI architecture can be designed that uses the scarce resources more effectively than CISC. We also expect this approach to reduce design time, the number of design errors, and the execution time of individual instructions.

Our initial version of such a computer is called RISC I. To meet our goals of simplicity and effective single-chip implementation, we placed the following "constraints" on the architecture:

1. Execute one instruction per cycle. RISC I instructions should be about as fast as, and no more complicated than, micro instructions in current machines such as PDP-11 or VAX. Furthermore, this simplicity makes microcode control unnecessary. Skipping this extra level of interpretation appears to enhance performance while reducing chip size.

2. All instructions are the same size. This again simplifies implementation. We intentionally postponed attempts to reduce program size.

3. Only load and store instructions access memory; the rest operate between registers. This restriction simplifies the design. The lack of complex addressing modes also makes it easier to restart instructions.

4. Support high-level languages (HLL). An explanation of the degree of support follows. Our intention is always to use high-level languages with RISC I.

RISC I supports 32-bit addresses, 8-, 16-, and 32-bit data, and several 32-bit registers. We intend to

examine support for operating systems and floating-point calculations in successors to RISC I.

It would appear that such constraints would result in a machine with substantially poorer code density or poorer performance or both. In spite of these constraints, the resulting architecture competes favorably with other state-of-the-art machines such as VAX 11/780. This is largely because of an innovative new scheme of register organization we call *overlapped register windows.*

SUPPORT FOR HIGH-LEVEL LANGUAGES

Clearly, new architectures should be designed with the needs of high-level language programming in mind. It should not matter whether a high-level language system is implemented mostly by hardware or mostly by software, provided the system hides any lower levels from the programmer.[7] Given this framework, the role of the architect is to build a cost-effective system by deciding what pieces of the system should be in hardware and what pieces should be in software.

The selection of languages for consideration in RISC I was influenced by our environment; we chose C and Pascal languages, because there is a larger user community and considerable local expertise. Given the limited number of transistors that can be integrated into a single-chip computer, most of the pieces of a RISC high-level language system are in software, with hardware support for only the most time-consuming events.

To determine what constructs are used most frequently and, if possible, what constructs use the most time in average programs, we looked first at the frequency of classes of variables in high-level language programs. Figure 1 shows data collected by Goldwasser for Pascal language[8] and by Cohen and Soiffer for C language.[9]

The most important observation was that integer constants appeared almost as frequently as components of arrays or structures. What is not shown is that over 80% of the scalars were local variables and over 90% of the arrays or structures were global variables.

We also looked at the relative dynamic frequency of high-level language statements for the same eight programs; the ones with averages over 1% are shown in Figure 2. This information does not tell what statements use the most time in the execution of typical programs. To answer that question, we looked at the code produced by typical versions of each of these statements. A "typical" version of each statement was supplied by W. Wulf (private communication, Nov. 1980) as part of his study on judging the quality of compilers. We used C compilers for VAX, PDP-11, and 68000 to determine the average number of instructions and memory references. By multiplying the frequency of occurrence of each statement with the corresponding number of machine instructions and memory references, we obtained the data shown in Figure 3, which is ordered by memory references.

The data in these tables suggests that the procedure CALL/return is the most time-consuming operation in typical high-level language programs. The statistics on operands emphasizes the importance of local variables and constants. RISC I attempts to make each of these constructs efficient, implementing the less-frequent operations with subroutines.

BASIC ARCHITECTURE OF RISC I

The RISC I instruction set contains a few simple operations (arithmetic, logical, and shift) that operate on registers. Instructions, data, addresses, and registers are 32 bits. RISC instructions fall into four categories (Figure 4): arithmetic-logical (ALU), memory access, branch, and miscellaneous. The execution time of a RISC I cycle is given by the time it takes to read a register, perform an ALU operation, and store the result back into a register. Register 0, which always contains 0, allows us to synthesize a variety of operations and addressing modes.

Load and store instructions move data between registers and memory. These instructions use two CPU cycles. We decided to make an exception to our constraint of single-cycle execution rather than to extend the general cycle to permit a complete memory access. There are eight variations of memory access instructions to accommodate sign-extended or zero-extended 8-bit, 16-bit, and 32-bit data. Although there appears to be only one addressing mode, *index plus displacement, absolute* and *register indirect* addressing can be synthesized using register 0 (Figure 5). (Using one register to always contain 0 dates back at least to CDC-6600 in 1964. It has also appeared in more recent designs.[10])

Branch instructions include CALL, return, conditional and unconditional jump. The conditional instructions are the standard set used originally in PDP-11 and are found in most 16-bit microprocessors today. Most of the

innovative features of RISC are found in CALL, return, and jump; they will be discussed in subsequent sections.

Figure 6 shows the 32-bit format used by register-to-register instructions and memory access instructions. For register-to-register instructions, DEST selects one of the 32 registers as the destination of the result of the operation, which itself is performed on the registers specified by SOURCE1 and SOURCE2. If IMM equals 0, the low-order 5 bits of SOURCE2 specify another register; if IMM equals 1, SOURCE2 expresses a sign-extended, 13-bit constant. Because of the frequency of occurrence of integer constants in high-level language programs, the immediate field has been made an option in every instruction. SCC determines if the condition codes are set. Memory access instructions use SOURCE1 to specify the index register and SOURCE2 to specify the offset. One other format, which combines the last three fields to form a 19-bit PC-relative address, is used primarily by the branch instructions.

Although comparative measurements of benchmarks are the real test of effectiveness, the examples in Figure 5 show that many of the important VAX instructions can be synthesized from simple RISC addressing modes and operation codes. Remember that register 0 (*r*0) always contains 0; specifying *r*0 as a destination does not change its value.

Register Windows

The previously mentioned investigations on using high-level languages indicate that the procedure CALL may be the most time-consuming operation in typical high-level language programs. Potentially, RISC programs may have an even larger number of calls, because the complex instructions found in CISCs are subroutines in RISC. Thus, the procedure CALL must be as fast as possible, perhaps no longer than a few jumps. The RISC *register window* scheme comes close to this goal. At the same time, this scheme also reduces the number of accesses to data memory.

Using procedures involves two groups of time-consuming operations: saving or restoring registers on each CALL or return, and passing parameters and results to and from the procedure. Because our measurements on high-level language programs indicate that local scalars are the most frequent operands, we wanted to support the allocation of locals in registers. Baskett[11] and Sites[12] suggested that microprocessors keep multiple banks of registers on the chip to avoid register saving and restoring. Thus, each procedure CALL results in a new set of registers being allocated for use by that new procedure. The return just alters a pointer, which restores the old set. A similar scheme was adopted by RISC I; however, some of the registers are not saved or restored on each procedure CALL. These registers (*r*0 through *r*9) are called *global* registers.

In addition, the sets of registers used by different processes are overlapped to allow parameters to be passed in registers. In other machines, parameters are usually passed on the stack with the calling procedure using a register (frame pointer) to point to the beginning of the parameters (and also to the end of the locals). Thus, all references to parameters are indexed references to memory. Our approach is to break the set of window registers (*r*10 to *r*31) into three parts (Figure 7). Registers 26 through 31 (HIGH) contain parameters passed from "above" the current procedure; that is, the calling procedure. Registers 16 through 25 (LOCAL) are used for the local scalar storage exactly as described previously. Registers 10 through 15 (LOW) are used for local storage and for parameters passed to the procedure "below" the current procedure (the called procedure). On each procedure CALL, a new set of registers, *r*10 to *r*31, is allocated; however, we want the LOW registers of the "caller" to become the HIGH registers of the "callee." This is accomplished by having the hardware overlap the LOW registers of the calling frame with the HIGH registers of the called frame; thus, without moving information, parameters in registers 10 through 15 appear in registers 25 through 31 in the called frame. Figure 8 illustrates this approach for the case in which procedure *A* calls procedure *B*, which calls procedure *C*.

Multiple register banks require a mechanism to handle the case in which there are no free register banks available. RISC I handles this with a separate register overflow stack in memory and a stack pointer to it. Overflow and underflow are handled with a trap to a software routine that adjusts that stack. Because this routine can save or restore several sets of registers, the overflow/underflow frequency is based on the local variations in the depth of the stack rather than on the absolute depth The effectiveness of this scheme depends on the relative frequency of overflows and underflows; studies by Halbert and Kessler[13] indicate that overflow will occur in less than 1% of the calls with only 4 to 8 register banks. (Other machines, such as BBN C/70, contain register banks, but they do not overlap their windows.)

The final step in allocating variables in registers is handling the problem of pointers. Pointers to variables require that variables have addresses. Because registers do not normally have addresses, one could let the compiler determine what variables have pointers and put such variables in memory. This precludes separate compilation, slows down access to these variables, and is beyond state-of-the-art compiler technology found in most companies and universities. RISC I solves that problem by giving addresses to the window registers. If we reserve a portion of the address space, we can determine, with one comparision, whether an address points to a register or to memory. Because the only instructions to access memory are load and store, and they take an extra cycle already, we can add this feature without reducing the performance of the load and store instructions. This permits the use of straightforward compiler technology and still leaves a large fraction of the variables in registers.

Delayed Jump

The normal RISC I instruction cycle is just long enough to execute the following sequence of operations:

1. Read a register
2. Perform an ALU operation
3. Store the result back into a register

We increase performance by prefetching the next instruction duing the execution of the current instruction. This introduces difficulties with branch instructions. Several high-end machines have elaborate techniques to prefetch the appropriate instruction after the branch,[14] but these techniques are too complicated for a single-chip RISC. Our solution was to redefine jumps so that they do not take effect until after the following instruction; we refer to this as the *delayed jump*. (This approach to branching dates back to MANIAC I in 1952 and is now commonly used in microprogramming.)

The delayed jump allows RISC I always to prefetch the next instruction during the execution of the current instruction. The machine language code is suitably arranged so that the desired results are obtained. Because RISC I is always intended to be programmed in high-level languages, we will not "burden" the programmer with this complexity; the burden will be carried by the programmers of the compiler, the optimizer, and the debugger.

To illustrate how the delayed branch works, Figure 9a shows a sequence of instructions, which, in machines with normal jumps, would be executed in the order 100, 101, 102, 105, ... To get that same effect in RISC I, we would have to insert NOP (Figure 9b). In this case, the sequence of instructions for RISC I is 100, 101, 102, 103, 106, ... In the worst case, every jump could take two instructions. The RISC I software, however, includes an optimizer that tries to rearrange the sequence of instructions to perform the equivalent operations without NOP. Such an optimized RISC I sequence is 100, 101, 102, 105, ... (Figure 9c). Because the instruction following a jump is always executed, and the jump at 101 is not dependent on the ADD at 102, this sequence is equivalent to the original program segment in Figure 9a.

EVALUATION

We will now evaluate the register window scheme, the delayed branch, and the overall performance of RISC I.

Register Windows

The results of running two benchmarks have shown that the window registers have been effective in reducing the cost of using procedures. The puzzle and quicksort programs, discussed below, are highly recursive routines. Figure 10 shows the maximum depth of recursion, the number of register window overflows and underflows, and the total number of words transferred between memory and the RISC CPU as a result of the overflows and underflows. It also shows the memory traffic caused by saving and restoring registers in VAX. For this simulation, we assumed that half of the registers were saved on an overflow and half were restored on an underflow. We found that for RISC I, an average 0.37 words were transferred to memory per procedure invocation for the puzzle program and 0.07 for quicksort. Note that half of the data memory references in quicksort were the result of the CALL/return overhead of VAX.

We also compared the performance of the RISC I procedure mechanism to that of more traditional machines. We chose VAX, PDP-11, and M68000 as representatives of modern computers. Figure 11 shows the numbers of instructions, their total sizes in bytes, and the numbers of register accesses and data memory accesses for these three computers and for RISC I. The data was collected by looking at the code generated by C compilers for these four machines for procedure CALL

and return statements, assuming that two parameters are passed and requiring that 3 registers must be saved. It appears that this scheme reduces the cost of using procedures significantly.

This scheme also reduces off-chip memory accesses. In traditional machines, generally 30% to 50% of the instructions access data memory, with not more than 20% of the instructions being register-to-register.[15] Because RISC I arithmetic and logical instructions cannot access memory, it might be expected that even a higher fraction of the instructions would be data transfer. This was not the case. The static frequencies of RISC I instructions for nine typical C programs show that less than 20% of the instructions were loads and stores, and more than 50% of the instructions were register-to-register. RISC I has successfully changed the allocation of variables from memory into registers. This indicates that RISC I requires a lower number of the slower off-chip memory accesses. It also indicates that complex addressing modes are not necessary to obtain an effective machine.

Delayed Jump

The performance of our scheme can be evaluated by counting the number of NOP instructions in a program. Static figures before optimization show that in typical C programs, about 18% of the instructions are NOP instructions inserted after jump instructions. A simple peephole optimizer built by students reduced this to about 8%. The optimizer did well on unconditional branches (removing about 90% of NOP instructions), but not so well with conditional branches (removing only about 20% of NOP instructions). This optimizer was improved to replace NOP by the instruction at the target of a jump. This technique can be applied to conditional branches if the optimizer determines that the target instruction modifies temporary resources; for example, an instruction that only modifies the condition codes. In quicksort, this removes all NOP instructions except those that follow return instructions. The dynamic effectiveness of the delayed branch must now include the number of NOP instructions plus the number of instructions after conditional branches that need not be executed for a particular jump condition. The total percentages of either type of instruction for three programs discussed below are 7%, 22%, and 4%.

Overall Performance

To judge the effectiveness of the RISC I architecture, we compared it with VAX, because it is an efficient and a popular modern machine, and PDP-11, because it was the first machine with a C compiler and many persons assume that it is an ideal C machine. (This assumption is not valid. Although the development of C language was somewhat influenced by the architecture of PDP-11, most features of C came from B language, which was an interpreted language not tailored to any architecture.) Figure 12 and 13 compare the static numbers of instructions and the static sizes for 11 typical C programs for the three machines. The compilers used are similar: the VAX and RISC C compilers are both based on the UNIX portable C compiler[16] the compiler for PDP-11 is based on the Ritchie C compiler.[17] Experiments comparing the Ritchie and Portable C compilers for PDP-11 have shown that the average difference in the size of generated code is within 1% (S. C. Johnson, private communication, Feb. 1981).

We found that on the average, RISC uses only two-thirds more instructions than VAX and about two-fifths more than PDP-11, in spite of the fact that RISC I has simple instructions and addressing modes. The most surprising result was that the RISC programs were only about 50% larger than the programs for the other machines even though size optimization was virtually ignored.

Our main goal for RISC I was to obtain good performance; thus dynamic results are the most interesting. We used a C program developed by F. Baskett (private communication, Nov. 1980) called "puzzle." This program is essentially a recursive bin-packing program that solves a three-dimensional puzzle. It displays many features of typical programs, except that there are less than 0.2% procedure calls, the call stack gets deep (20 nested procedure calls), and there are a relatively large number of loops. There are several versions of this program. Version A, which we received from Baskett, accesses arrays with subscripts and does not declare register variables. (Register variables are hints, supplied by the programmer, to the C compiler that this variable will be used frequently and should be kept in a register). We produced version B by converting some local variables into register variables. In version C, we changed the way arrays are accessed from using subscripts to using pointers. The dynamic information about each version of this program is shown in Figures 14 and 15 . The statistics of VAX came from an instruction trace program developed by Henry.[18]

RISC I statistics came from a simulator developed by Tamir.

The results of running the recursive quicksort program are also shown in Figure 14. This program sorts 2,600 fixed-length character strings. The only unusual feature of this program is that it has relatively more memory references than most programs. The execution of this program results in 1,713 multiply operations and 1,712 divide operations, which are subroutines in RISC I.

There is much important information in Figure 14. The first is that it made no difference to RISC whether we used version A or B of the puzzle program. This is because the architecture makes it relatively simple for a compiler to allocate local scalars in registers, so there is no need for a language to give hints telling which should be used. Thus, a one-pass Pascal compiler, which does not normally allocate registers for machines like VAX, would likely allocate variables in registers for RISC I and, therefore, result in the same relative memory traffic as version A of the puzzle program.

Note that most commercial compilers do little optimization. For example, even a three-pass, optimizing Pascal compiler for DEC 10 does not allocate locals or parameters in registers.[19] It is unreasonable for architects to expect, in the near future, sophisticated optimization from production quality compilers.

RISC I was successful in reducing the number of data accesses substantially in all programs. The number of instruction words accessed, however, increased. This is because of the number of NOP instructions executed and the inefficient encoding of RISC I instructions. We expect that successors to RISC I could reduce this difference.

The final, and perhaps most important, figure of merit is execution time. This was easy to determine for VAX 11/780, but difficult for RISC I as we do not have any hardware. Our execution time was based on low-level circuit simulations of early RISC I designs. Using student circuit designers, we estimated that a RISC cycle is 400 nsec: 100 nsec to read one of 135 registers, 200 nsec to perform a 32-bit addition, and 100 nsec to store the result in one of 135 registers. We can argue that this is both optimistic and pessimistic: it is optimistic because it is unlikely that students can successfully build something that fast in their first pass, and it is pessimistic because it is likely that an experienced IC design team could build a much faster machine. Nevertheless, the student-technology single-chip RISC I may still be faster than VAX 11/780 for all benchmarks mentioned previously.

We must mention that although our results are encouraging, they are estimates based upon simulations of only two programs. Further benchmarks must be finished before we can accurately characterize the performance of RISC I.

MEMORY INTERFACE

In most computers, the interface to memory is a main performance bottleneck, so this point must be given special consideration. In our discussions and simulations, we assumed that we can access main memory in a single RISC CPU cycle. Depending on the assumptions that we make for our CPU cycle time, and the size of the main memory, this assumption may be too optimistic. We thus reworked our benchmarks also under the assumption that two CPU cycles are required to access data memory. Performance degraded only 10%, because the register window scheme reduces the number of off-chip data references. Data references do not constitute a problem, but allowing two cycles to fetch instructions out of memory would reduce performance by almost a factor of 2.

Clearly, this memory interface will be an increasingly critical point as the intrinsic speed of CPU increases with technologic advances. Accesses to memory can be forced to come mainly from on-chip, either with a large register file or with an on-chip cache and associated memory hierarchy.[6]

An on-chip cache would be beneficial for RISC. It is sometimes forgotten that a cache is ineffective if it is too small. In our opinion, an effective data cache would have to be quite a bit larger than our planned register file, especially if it was to provide the same number of ports as the register file. More-complicated translation and decoding might even strech the basic CPU cycle time. Given the limited amount of circuitry we can place onto a chip at this point, and given the university environment and our student designers, a register file is clearly the safer way to go.

Although the problem of data accesses has been alleviated by the large number of registers and the effective window scheme, the number of instruction fetches has actually increased because of the simplicity of individual instructions. Instruction fetches from main memory are indeed a major speed-limiting factor. An instruction cache is a desirable commodity. Because

there is no need for CPU to write into this cache, its controller can be simpler than that of a data cache. We decided that RISC I should not be burdened with the design of a full-blown on-chip cache, but an instruction cache would definitely be a good idea for the next-generation RISC.

SUMMARY

From our limited experience based on the results of a few small programs, it appears that the reduced instruction set computer is a promising style of computer design. We have convinced ourselves that complicated addressing schemes are not a vital part of high-throughput machines. The register window scheme appears to make significant contributions toward the performance of our architecture and should be seriously considered in other machines.

We have taken out most of the complexity of modern computers without sacrificing much in code density while improving performance. The loss of complexity has not reduced the functionality of RISC; the chosen subset, especially when combined with the register window scheme, emulates more complex machines. It also appears we can build a single-chip computer much sooner than the traditional architectures. We are encouraged by these results and have begun the design of a single-chip RISC I as part of a multiterm class project.

ACKNOWLEDGMENTS

This research was sponsored 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.

The RISC Project has been sustained by a large number of students. We would like to thank all those in the Berkeley community who have helped to push RISC from a concept to an engineering experiment. The contributions of the following persons were important to RISC: C statistics by E. Cohen and N. Soiffer; Pascal statistics by S. Goldwasser; C compiler initially by D. Doucette and K. Shoens with extensive revisions by R. Campbell; RISC 0 optimizer by D. Fitzpatrick; RISC I optimizer by R. Campbell; assembler by R. Campbell and later revised by Y. Tamir; RISC 0 simulator by R. Campbell, E. Lock, and M. Hakam; RISC I simulator by Y. Tamir; ISPS description by G. Corcoran; window scheme based on an idea of F. Baskett, but designed by D. Halbert and P. Kessler; and LSI timing and suggested LSI implementation by M. Katevenis. We would also like to thank L. Dickman, D. Ditzel, R. Hyerle, M. Katevenis, J. Ousterhout, D. Presotto, D. Ungar, and K. Van Dyke for their suggestions on this paper.

REFERENCES

[1]W. D. Strecker. VAX-11/780: A virtual address extension to the DEC PDP-11 family, *Proceedings of NCC* (June 1978), 967-980.

[2]B. G. Utley et al. In *IBM System/38 Technical Developments* (GS80-0237), 1978, 1-110

[3]S. Colley et al. The object-based architecture of the Intel 432, *COMPCON* (Feb. 1981).

[4]D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer, *Computer Architecture News*, 8 (15 Oct. 1980), 25-33.

[5]D. A. Patterson, E. S. Fehr, and C.H. Séquin. Design considerations for the VLSI processor of X-tree. *The 6th Annual International Symposium on Computer Architecture* (April 1979).

[6]D. A. Patterson and C. H. Séquin. Design considerations for single-chip computers of the future, *IEEE Journal of Solid-State Circuits*, SC-15 (Feb. 1980), 44-52; and *IEEE Transactions on Computers*, C-29 (Feb. 1980), 108-116. (Joint special issue on microprocessors and microcomputers.)

[7]D. R. Ditzel and D. A. Patterson. Retrospective on high-level language computer architecture, *The 7th Annual International Symposium on Computer Architecture* (May 1980), 97-104.

[8]S. Goldwasser. Dynamic Pascal statistics (in progress, Sept. 1980).

[9]E. Cohen and N. Soiffer. Static and dynamic statistics of C "CS 292R Final Reports" (University of California at Berkeley, 1980), 101-140.

[10]S. C. Johnson. A 32-bit processor design (Computer science technical report No. 80), Bell Laboratories, 1979.

[11]F. Baskett. A VLSI Pascal machine (Public lecture), University of California, 1978.

[12]R. L. Sites. How to use 1000 registers, *Caltech Conference on VLSI* (Jan. 1979).

[13]D. Halbert and P. Kessler. Windows of overlapping register frames, "CS 292R Final Reports" (University of California at Berkely, 1980), 82-100.

[14]D. Morris and R. N. Ibbett. *The MU-5 Computer System* (Springer-Verläg, 1979).

[15]W. C. Alexander and D. B. Wortman. Static and dynamic characteristics of XPL programs, *Computer*, 8 (Nov. 1975), 41-46.

[16]S. C. Johnson. A portable compiler: Theory and practice, *Proceedings of the Fifth Annual ACM Symposium of Programming Languages (Jan. 1978), 97-104.*

[17]D. M. Ritchie. A tour through the UNIX C compiler (Unpublished), 1975.

[18]R. R. Henry. Techniques to measure static and dynamic operator and operand statistics on the VAX, (Unpublished report), University of California at Berkeley, 1980.

[19]R. N. Faiman and A. A. Kortesoja. An optimizing Pascal compiler, *IEEE Transactions of Software Engineering*, (Nov. 1980), 512-519.

	C				Pascal				
	C1	C2	C3	C4	P1	P2	P3	P4	Ave
Integer Constant	25	11	29	28	11	16	8	16	18 ± 8
Scalar	37	45	66	62	70	72	62	63	60 ± 12
Array/Structure	36	43	5	10	19	12	30	20	22 ± 13

C1 PCC - The Portable C Compiler for the VAX
C2 CIFPLOT - a program that plots VLSI mask layouts on a dot plotter
C3 NROFF - a text formatting program
C4 SORT - the UNIX sorting program
P1 COMP - A Pascal P-code style compiler
P2 MACRO - The macro expansion phase of the SCALD I design system
P3 PRINT - A prettyprinter for Pascal
P4 DIFF - A program that finds the differences between two files

Figure 1. Dynamic Percentage of Operands in C and Pascal

	Pascal				
statements†	P1	P2	P3	P4	AVERAGE
assign	32	42	29	40	36 ± 5
begin	18	19	18	25	20 ± 3
if	29	24	30	12	24 ± 7
call	12	11	13	11	12 ± 1
with	2	0	4	10	4 ± 4
loop	4	4	4	3	4 ± 0
case	3	0	1	0	1 ± 1
	C				
statements	C1	C2	C3	C4	AVERAGE
assign	22	50	25	56	38 ± 15
if	59	31	61	22	43 ± 17
call	6	17	9	16	12 ± 5
loop	2	2	3	5	3 ± 1
goto	9	0	1	1	3 ± 4
case	2	-	-	0	<1 ± 1

Figure 2. Relative Frequency of Pascal and C Statements
(Because statements can be nested, we count each occurrence of a statement. Loop statements are counted once per execution rather than once per iteration. For example, if two IF statements and three assignment statements appear in a loop that iterates 5 times, we would count 26 statements with 15 assignments, 10 IF statements, and one loop. The BEGIN statement is counted only if it allocates local variables. The WITH statement qualifies a record name.)

statements	HLL (# occurrence)		WEIGHTED (# instr.)		WEIGHTED (# mem. ref.)	
HLL	P	C	P	C	P	C
call/return	12±1	12±5	30±3	33±14	43±4	45±19
loops	4±0	3±1	40±3	32±6	32±2	26±5
assign	36±5	38±15	12±2	13±5	14±2	15±6
if	24±7	43±17	11±3	21±8	7±2	13±5
begin	20±1	-	5±0	-	2±0	-
with	4±1	-	1±0	-	1±0	-
case	1±1	<1±1	1±1	1±1	1±1	1±1
goto	-	3±1	-	0±0	-	0±0

Figure 3. Weighted Relative Frequency of HLL Statements, Ordered by Memory References.
(For the CALL statement, we counted passing parameters, saving/restoring general registers, and saving/restoring the program counter. The IF and CASE statements include instructions to evaluate expressions and to jump. For loop statements, we count all machine instructions executed during each iteration.)

Instruction	Operands	Comments	
ADD	S1,S2,Rd	Rd ← S1 + S2	integer add
ADDC	S1,S2,Rd	Rd ← S1 + S2 + carry	add with carry
SUB	S1,S2,Rd	Rd ← S1 - S2	integer subtract
SUBC	S1,S2,Rd	Rd ← S1 - S2 - carry	subtract with carry
AND	S1,S2,Rd	Rd ← S1 & S2	logical AND
OR	S1,S2,Rd	Rd ← S1 | S2	logical OR
XOR	S1,S2,Rd	Rd ← S1 xor S2	logical EXCLUSIVE OR
SLA	S1,S2,Rd	Rd ← S1 shifted by S2	shift left arithmetic
SRA	S1,S2,Rd	Rd ← S1 shifted by S2	shift right arithmetic
SLL	S1,S2,Rd	Rd ← S1 shifted by S2	shift left logical
SRL	S1,S2,Rd	Rd ← S1 shifted by S2	shift right logical
LDL	(Rx)X,Rd	Rd ← M[Rx+X]	load long
LDSU	(Rx)X,Rd	Rd ← M[Rx+X]	load short unsigned
LDSS	(Rx)X,Rd	Rd ← M[Rx+X]	load short signed
LDBU	(Rx)X,Rd	Rd ← M[Rx+X]	load byte unsigned
LDBS	(Rx)X,Rd	Rd ← M[Rx+X]	load byte signed
STL	Rm,(Rx)X	M[Rx+X] ← Rm	store long
STS	Rm,(Rx)X	M[Rx+X] ← Rm	store short
STB	Rm,(Rx)X	M[Rx+X] ← Rm	store byte
JMP	COND,X(Rm)	pc ← X+Rm	conditional jump
JMPR	COND,Y	pc ← pc + Y	conditional relative
CALL	Rm,X(Rn)	Rm ← pc, next pc ← X+Rn, CWP–	
CALLR	Rm,Y	Rm ← pc, next pc ← pc + Y, CWP–	
RET	Rm,X	pc ← Rm+X, CWP++	
GTLPC	Rm	Rm ← last pc	get last pc
GTIN	Rm	Rm ← INR	get interrupt number

Figure 4. Assembly Language Definition for RISC
(SUB and SUBC really represent two operation codes each, because the operations are not commutative.)

Addressing	VAX	RISC equivalent
Register	Rn	Rn
Immediate	#literal	#literal
Indexed	Rx + displ	Rx + displ
Absolute	@#address	r0 + displ
Reg Indirect	(Rx)	Rx + 0

Operation	VAX		RISC equivalent	
Compare	cmpl	Rm,Rn	sub	Rm,Rn,r0,{c}
Reg-Reg Move	movl	Rm,Rn	add	r0,Rm,Rn
Compare to 0	tstl	Rn	sub	Rn,r0,r0,{c}
	tstl	A	ldl	(r0)A,r0,{c}
Clear	clrl	Rn	add	r0,r0,Rn
	clrl	A	stl	r0,(r0)A
Two's Complement	mnegl	Rm,Rn	sub	r0,Rm,Rn
One's Complement	mcoml	Rm,Rn	xor	Rm,#-1,Rn
Load Const	movl	$N,Rm	add	r0,#N,Rm
Increment	incl	Rn	add	Rn,#1,Rn
Decrement	decl	Rn	sub	Rn,#1,Rn
Check index bounds, (A[0:U])	index	Rm,#0,#U, #1,A,Rn;	sub	Rm,#U,r0{c};
			jmp	lequ,OK;
trap if error, & read A[Rm]	movb	(Rn),Rp	call	error;
			OK: ldbu	(Rm)A,Rp

Figure 5. Synthesizing VAX Instructions
(The approach to bounds checking shown in the last example is better than the normal algorithm. We can think of an index as an unsigned integer because 0 ≤ index ≤ *U*. A twos complement negative number (1*X*...*X*) is then a very large unsigned number, so we only need to make one unsigned test instead of two signed tests. Nonzero lower bounds are handled by repeating the sequence and including a multiply and an add. This idea resulted from a discussion between B. Joy, P. Kessler, and G. Taylor. Taylor coded the examples and found that on VAX 11/780, the sequence of simple instructions was always faster than the index instruction.)

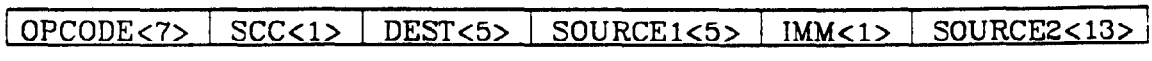

Figure 6. RISC I Basic Instruction Format

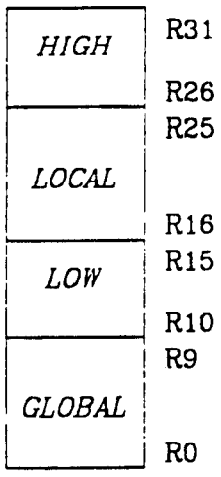

Figure 7. Naming Within a Virtual RISC I Register Window

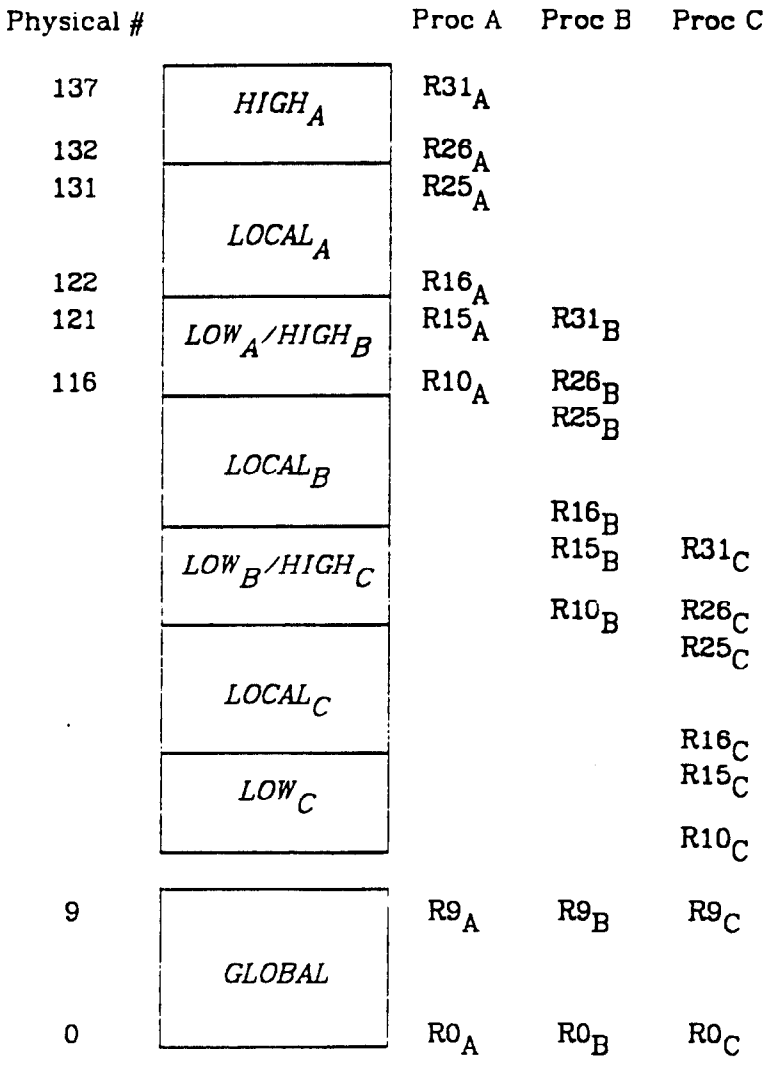

Figure 8. Usage of Three Overlapped Register Windows

Address	(a) Normal Jump		(b) Delayed Jump		(c) Optimized Delayed Jump	
100	LOAD	X,A	LOAD	X,A	LOAD	X,A
101	ADD	1,A	ADD	1,A	**JUMP**	**105**
102	**JUMP**	105	**JUMP**	**106**	**ADD**	**1,A**
103	ADD	A,B	**NOP**		ADD	A,B
104	SUB	C,B	ADD	A,B	SUB	C,B
105	**STORE**	**A,Z**	SUB	C,B	**STORE**	**A,Z**
106			**STORE**	**A,Z**		

Figure 9. Normal and Delayed Jumps

	Calls + Returns % instrs	Maximum Nested Depth	RISC I overflows+ underflows	Data Memory Traffic RISC I # words	Data Memory Traffic VAX # words
puzzle	43k	20	124	8k	444k
	0.7%			0.8%	28.0%
quicksort	111k	10	64	4k	696k
	8.0%			1.0%	50.0%

Figure 10. Memory Traffic Caused by CALL/Return
(These are the results of the pointer version of puzzle. The subscripted versions, A and B, use 235K words and 363K words, respectively.)

	Instructions Executed	Size (bytes)	Register accesses	Data Memory accesses
VAX 11	5	16	59	19
68000	9	30	41	12
PDP-11	19	44	51	15
RISC I	6	24	12	0.2

Figure 11. Procedure CALL/Return Overhead, Including Parameter Passing

Name	VAX	VAX rel	11/70	11/70 rel	RISC	RISC rel
acker	32	1.00	41	1.28	52	1.63
brelse	30	1.00	39	1.30	63	2.10
fun	9	1.00	15	1.67	12	1.33
qsort	101	1.00	159	1.57	161	1.59
stats	54	1.00	98	1.72	104	1.93
sym	43	1.00	72	1.67	83	1.93
towers	30	1.00	37	1.23	33	1.10
spell	774	1.00	985	1.27	1094	1.41
sort	1213	1.00	1395	1.15	1849	1.52
finger	1578	1.00	1961	1.24	2588	1.64
puzzle	381	1.00	496	1.30	617	1.62
Average	386	1.0 ± .0	482	1.4 ± .2	605	1.6 ± .3

Figure 12. Static Number of Instructions: Absolute and Ratio to VAX

Name	VAX	VAX rel	11/70	11/70 rel	RISC	RISC rel
acker	120	1.00	130	1.08	208	1.73
brelse	172	1.00	140	0.81	252	1.47
fun	32	1.00	44	1.38	48	1.50
qsort	436	1.00	462	1.06	644	1.48
stats	284	1.00	316	1.11	416	1.46
sym	204	1.00	220	1.08	332	1.63
towers	100	1.00	124	1.24	132	1.32
spell	2996	1.00	3106	1.04	4376	1.46
sort	4996	1.00	4582	0.92	7396	1.48
finger	6544	1.00	6490	0.99	10352	1.58
puzzle	1668	1.00	2004	1.13	2468	1.48
Average	1596	1.0 ± .0	1602	1.1 ± .1	2420	1.5 ± .1

Figure 13. Program Size: Number of Bytes and Ratio to VAX

	Puzzle (Subscripts)			*Puzzle (Pointers)*		*Quicksort*	
	A VAX	B VAX	A,B RISC	C VAX	C RISC	D VAX	D RISC
Time (secs)	11.3	9.5	5.2	4.0	3.6	1.8	.8
# Internal Cycles (M)	65	56	13	22	9	9	2.0
# Instr. Exec. (M)	10	8.2	11	5.3	7.2	1.0	1.6
# Instr. Words (M)	11	5.4	11	4	7.2	.8	1.6
# Data Mem. Access (M)	5.8	3.4	1.7	1.4	1	1.4	.4

Figure 14. Dynamic Statistics for C Programs for VAX and RISC
(The number of internal cycles is the number of micro instructions executed on VAX [= time/200 nsec] and the number of basic register-to-register cycles on RISC.)

	Puzzle (Subscripts)			*Puzzle (Pointers)*		*Quicksort*	
	A VAX	B VAX	A,B RISC	C VAX	C RISC	D VAX	D RISC
JUMP	2.52 25%	1.68 20%	1.73 17%	1.68 32%	1.73 24%	0.22 21%	0.23 14%
NOP	-	-	0.02 0.2%	-	0.67 9%	-	0.02 1.2%
CALL	0.02 0.2%	0.02 0.2%	0.02 0.2%	0.02 0.4%	0.02 0.3%	0.05 4.8%	0.06 3.7%
RET	0.02 0.2%	0.02 0.2%	0.02 0.2%	0.02 0.4%	0.02 0.3%	0.05 4.8%	0.06 3.7%
INC/DEC	0.80 8%	0.75 9%	-	0.00 0.0%	-	0.06 6%	-
ADD	1.53 15%	1.53 19%	3.32 33%	1.54 29%	2.47 35%	0.02 1.9%	0.48 29%
SUB	-	-	0.82 8%	-	0.85 12%	0.04 3.8%	0.25 15%
CMP	0.82 8%	0.76 9%	-	0.80 15%	-	0.10 10%	-
SHF	1.53 15%	1.53 19%	2.47 24%	0.06 1.1%	0.38 5%	0.00 0.1%	0.00 0.1%
STORE	-	-	0.04 0.4%	-	0.04 0.6%	-	0.15 9%
LOAD	-	-	1.67 17%	-	0.92 13%	-	0.24 15%
TEST(mem)	0.88 9%	0.88 11%	-	0.88 17%	-	0.00 0.1%	-
MOV	1.65 16%	0.82 10%	-	0.09 1.7%	-	0.33 31%	-
PUSH	0.04 0.4%	0.04 0.5%	-	0.04 0.8%	-	0.12 11%	-
MISC	0.20 2.0%	0.20 2.4%	-	0.20 3.8%	-	0.06 6%	0.14 9%
TOTAL	10.01 100%	8.23 100%	10.11 100%	5.33 100%	7.10 100%	1.05 100%	1.63 100%

Figure 15. Dynamic Instruction Mix for C Programs, Million Instructions