A VLSI RISC

by

D. A. Patterson and C. H. Sequin

Memorandum No. UCB/ERL M82/10

17 February 1982

A VLSI RISC

by

D. A. Patterson and C. H. Séquin

Memorandum No. UCB/ERL M82/10

17 February 1982

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A VLSI RISC†

David A. Patterson and Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

ABSTRACT

The Reduced Instruction Set Computer (RISC) Project investigates an alternative to the general trend toward computers with increasingly complex instruction sets: With the proper set of instructions and corresponding architecture, one can design a machine with high throughput. 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 takes less time to design.

This paper presents the architecture of RISC I and its 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, along with statistics on the 45,000 transistor VLSI implementation of RISC I. Although instructions are simpler, the average length of programs was found not to exceed those of more complicated machines by more than a factor of two. Preliminary benchmarks demonstrate the performance advantages of RISC I; it seems feasible to build a single-chip computer faster than the fastest minicomputers on the market today.

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, the VAX-11[2] to the PDP-11, the IBM System/38[3] to the System/3, and the Intel iAPX-432[4] to the 8086. We will call this class of computers Complex Instruction Set Computers (CISC). Some negative consequences of this complexity are increased design time, increased design errors, and inconsistent implementations.[5]

Investigations of VLSI architectures indicated that a major design limitation is the delay-power penalty of data transfers across chip boundaries and the still

† An earlier version of this paper, entitled "RISC I: A Reduced Instruction Set VLSI Computer," was presented at the *Eighth Annual Symposium on Computer Architecture*, May 1981, in Minneapolis, Minnesota.[1]

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 whether the extra hardware needed to implement a CISC is the best way to use these "scarce" resources.

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 one can design a suitable VLSI architecture that uses the scarce resources more effectively than a CISC. We also expect this approach to reduce design time, the 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 somewhat artificially placed the following design constraints on the architecture:

Execute one instruction per cycle. RISC I instructions should be about as fast and no more complicated than microinstructions in current machines such as the PDP-11 or VAX.

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

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.

Support High-Level Languages (HLL). The degree of support is explained below. Our intent is to optimize the performance of RISC I for use with high-level languages.

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 these constraints, imposed by our desire for simplicity and regularity, would result in a machine with substantially poorer code density, poorer performance, or both. In spite of these constraints, the resulting architecture competes favorably with other state-of-the-art microprocessors and minicomputers. This is due largely to a 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, however, 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 in software.

The selection of languages for consideration in RISC I was influenced by our environment; we chose 'C' since 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 first looked at the frequency of classes of variables in high-level language programs. Data

collected for Pascal and C are shown in Table 1.

Table 1.
Dynamic Percentage of Operands in C and Pascal †

	C				Pascal				
	C1	C2	C3	C4	P1	P2	P3	P4	Ave
Integer Constant	25	11	29	28	14	18	11	20	20 ± 7
Scalar	37	45	66	62	63	68	46	54	55 ± 11
Array/Structure	36	43	5	10	23	14	43	25	25 ± 14

†Program	Explanation
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

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

We also looked at the relative dynamic frequencies of high-level language statements for the same eight programs; average occurrences over 1% are shown in Table 2. This information does not tell what statements use the most time in the execution of typical programs. To answer that question, we have to look at the code produced by typical versions of each of these statements. A "typical" version of each statement was supplied by Wulf as part of his study into judging the quality of compilers. [8] We used C compilers for the VAX, PDP-11, and 68000 to determine the average number of instructions and memory references per statement. By multiplying the frequency of occurrence of each statement with the corresponding number of machine instructions and memory references we obtain Table 3, which is ordered by memory references.

The data in Table 3 suggests that the procedure call/return is the most time-consuming operation in typical high-level language programs. These results corroborate studies by Lunde[9] and Wichmann. [10] The statistics on operands found in Table 1 emphasize the importance of local variables and constants. RISC supports HLL by enhancing performance of the most time-consuming features of typical HLL programs, as opposed to making the architecture "close" to a particular HLL; thus RISC I attempts to handle local variables, constants, and procedure calls efficiently while leaving less frequent operations to instruction sequences or subroutines.

Table 2.
Relative Frequency of Pascal and C statements.

Pascal					
statements†	P1	P2	P3	P4	AVERAGE
assign	39	52	35	53	45 ± 8
if	35	30	36	16	29 ± 8
call	15	14	18	15	15 ± 1
with	2	0	5	13	5 ± 5
loop	5	5	5	4	5 ± 0
case	4	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

Table 3.
Weighted Relative Frequency of HLL Statements.
(ordered by memory references)

statements†	HLL (# occurrence)		WEIGHTED (# mach. instr.)		WEIGHTED (# mem. ref.)	
HLL	P	C	P	C	P	C
call/return	15±1	12±5	31±3	33±14	44±4	45±19
loops	5±0	3±1	42±3	32±6	33±2	26±5
assign	45±5	38±15	13±2	13±5	14±2	15±6
if	29±7	43±17	11±3	21±8	7±2	13±5
with	5±5	-	1±0	-	1±0	-
case	1±1	<1±1	1±1	1±1	1±1	1±1
goto	-	3±1	-	0±0	-	0±0

† Because statements can be nested, we count each occurrence of a statement. Loop statements are counted once per execution rather than once per loop 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 *with* statement qualifies a record name.

‡ 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 the machine instructions executed during each iteration.

BASIC ARCHITECTURE OF RISC I

The RISC I architecture has 31 instructions, most of which do simple ALU and shift operations on registers. As shown in Table 4, they have been grouped into four categories: arithmetic-logical, memory access, branch, and miscellaneous. Instructions, data, addresses, and registers are 32 bits. The execution time of a RISC I cycle is given by the time it takes to read and add two registers, and then store the result back into a register. The global register 0, which always contains zero, allows us to synthesize a variety of operations and addressing modes.

Table 4.
Assembly Language Definition for RISC

Instruction	Operands	Comments	
ADD	Rs,S2,Rd	Rd ← Rs + S2	integer add
ADDC	Rs,S2,Rd	Rd ← Rs + S2 + carry	add with carry
SUB	Rs,S2,Rd	Rd ← Rs - S2	integer subtract
SUBC	Rs,S2,Rd	Rd ← Rs - S2 - carry	subtract with carry
SUBR	Rs,S2,Rd	Rd ← S2 - Rs	integer subtract
SUBCR	Rs,S2,Rd	Rd ← S2 - Rs - carry	subtract with carry
AND	Rs,S2,Rd	Rd ← Rs & S2	logical AND
OR	Rs,S2,Rd	Rd ← Rs | S2	logical OR
XOR	Rs,S2,Rd	Rd ← Rs xor S2	logical EXCLUSIVE OR
SLL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift left
SRL	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right logical
SRA	Rs,S2,Rd	Rd ← Rs shifted by S2	shift right arithmetic
LDL	(Rx)S2,Rd	Rd ← M[Rx+S2]	load long
LDSU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short unsigned
LDSS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load short signed
LDBU	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte unsigned
LDBS	(Rx)S2,Rd	Rd ← M[Rx+S2]	load byte signed
STL	Rm,(Rx)S2	M[Rx+S2] ← Rm	store long
STS	Rm,(Rx)S2	M[Rx+S2] ← Rm	store short
STB	Rm,(Rx)S2	M[Rx+S2] ← Rm	store byte
JMP	COND,S2(Rx)	pc ← Rx+S2	conditional jump
JMPR	COND,Y	pc ← pc + Y	conditional relative
CALL	Rd,S2(Rx)	Rd ← pc, next	call
		pc ← Rx+S2, CWP--	and change window
CALLR	Rd,Y	Rd ← pc, next	call relative
		pc ← pc + Y, CWP--	and change window
RET	Rm,S2	pc ← Rm+S2, CWP++	return and change window
CALLINT	Rd	Rd ← last pc; next CWP--	disable interrupts
RETINT	Rm,S2	pc ← Rm+S2; next CWP++;	enable interrupts
LDHI	Rd,Y	Rd<31:13>←Y; Rd<12:0>←0	load immediate high
GTLPC	Rd	Rd ← last pc	to restart delayed jump
GETPSW	Rd	Rd ← PSW	load status word
PUTPSW	Rm	PSW ← Rm	set status word

Load and store instructions move data between registers and memory. Rather than lengthen the general cycle to permit a complete memory access, these instructions use two CPU cycles. 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 the *index plus*

displacement addressing mode in data transfer instructions, *absolute* and *register indirect* addressing can be synthesized using register 0 (See Table 5).

Table 5.
Synthesizing VAX Addressing Modes.

Addressing	VAX	RISC equivalent
Register	Rs	Rs
Immediate	#literal	S2 (13-bit literal)
Indexed	Rx + displ	Rx + S2 (13-bit displacement)
Absolute	@#address	r0 + S2 (r0 ≡ 0)
Reg Indirect	(Rx)	Rx + 0

Branch instructions include call, return, conditional and unconditional jump. The conditional instructions are the standard set used originally in the PDP-11 and 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 later.

Figure 1 shows the 32-bit format used by register-to-register instructions and memory access instructions.

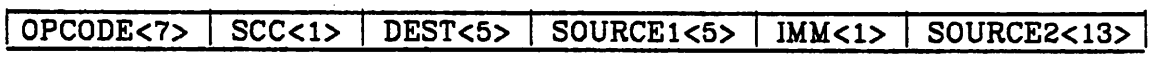

Figure 1. RISC I basic instruction format.

For register-to-register instructions, DEST selects one of the 32 registers as the destination of the result of the operation performed on the registers specified by SOURCE1 and SOURCE2. If IMM = 0, the low order five bits of SOURCE2 specify another register; if IMM = 1, SOURCE2 expresses a sign-extended 13-bit constant. As mentioned above, the frequency of integer constants in HLL programs suggests architectural support, so immediate operands are available in every instruction. SCC determines whether the condition codes are set. Memory access instructions use SOURCE1 to specify the index register and SOURCE2 to specify the offset. One other format combines the last three fields to form a 19-bit PC-relative address, and is used primarily by the branch instructions.

The examples in Table 6 show that many of the important VAX instructions can be synthesized from simple RISC addressing modes and opcodes. Comparative measurements of benchmarks will demonstrate the effectiveness of the chosen instruction set.

Table 6.
Synthesizing VAX Instructions.

Operation	VAX		RISC equivalent	
Reg-Reg Move	movl	Rm,Rn	add	r0,Rm,Rn (r0 ≡ 0)
Compare	cmpl	Rm,Rn	sub	Rm,Rn,r0,{c}
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(N < 2^{12})	add	r0,#N,Rm
	movl	$N,Rm(N ≥ 2^{12})	ldhi	#N<31:13>,Rm
			add	r0,#N<12:0>,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};
trap if error,			jmp	lequ,OK;‡
& read A[Rm]	movb	(Rn),Rp	call	error;
			OK: ldbu	(Rm)A,Rp

Register Windows

The previously mentioned investigations of the use of high-level languages suggest that the procedure call is the most time-consuming operation in typical high-level language programs. Potentially, RISC programs may have even more calls, because the complex instructions found in CISC's are subroutines in RISC. Thus, the procedure call must be as fast as possible, perhaps no longer than a few jumps. Because of the *register window* scheme, RISC I comes close to this goal. At the same time, this scheme also reduces data memory traffic.

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. As mentioned above, the frequency of local scalar variables justifies architectural support by placing locals in registers. Baskett[11] and Sites[12] proposed that microprocessors keep multiple banks of registers on the chip to avoid register saving and restoring. A similar scheme was adopted by RISC I; each procedure call results in the allocation of a new 'window' of registers from the large register file for use by that new procedure. The return just resets a pointer, restoring the old set. In addition, some of the registers are not saved or restored on each procedure call. These registers (r0 through r9) are called *global* registers.

‡ This approach is better than the normal algorithm. We can think of an index as an unsigned integer since 0 ≤ index ≤ U. A two's complement negative number (1X...X) is then a large unsigned number, so we only need make one unsigned test instead of two signed tests. Non-zero lower bounds are handled by subtracting the lower bound from the index, and multiple indices are handled by repeating the sequence and including a multiply and an add. This idea resulted from a discussion between Bill Joy, Peter Kessler, and George Taylor. Taylor coded the examples and found that on the VAX-11/780, the sequence of simple instructions was always faster than the index instruction. This optimization is found in the UNIX C optimizer.

Furthermore, the sets of registers used by different procedures 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 the end of the locals); thus, all references to parameters are indexed references to memory. Our approach is to partition the set of window registers (10-31) into the three parts defined by their respective overlap. Every procedure sees the set of registers shown in Figure 2.

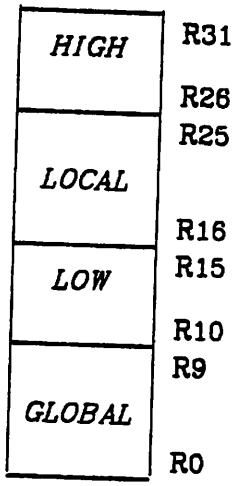

Figure 2. Naming within one Virtual RISC I Register Window.

Registers 26 through 31 (*HIGH*) contain parameters passed from "above" the current procedure; that is, from the calling procedure. Registers 16 through 25 (*LOCAL*) are used for local scalar storage. Registers 10 through 15 (*LOW*) are used for temporaries and parameters passed to the procedure "below" the current procedure (the called procedure). On each procedure call a new set of registers, numbered 10-31, is allocated. The LOW registers of the "caller" become the HIGH registers of the "callee" because of the hardware overlap between subsequent register windows. Thus, without moving information, parameters in registers 10-15 appear in registers 25-31 of the called window. Figure 3 illustrates this approach for the case where procedure *A* calls procedure *B*, which calls procedure *C*.

If the nesting depth is sufficiently large, all register windows will be used. RISC I handles such an overflow with a separate stack in memory. 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] show that with eight register banks overflow will occur in less than 1% of the calls. This suggests that programs exhibit locality in the dynamic nesting of procedures just as they exhibit locality in memory references.

Another problem with variables in registers occurs when we want to reference them with pointers. This requires that these variables have addresses. Because registers normally do not have memory addresses, one could let the

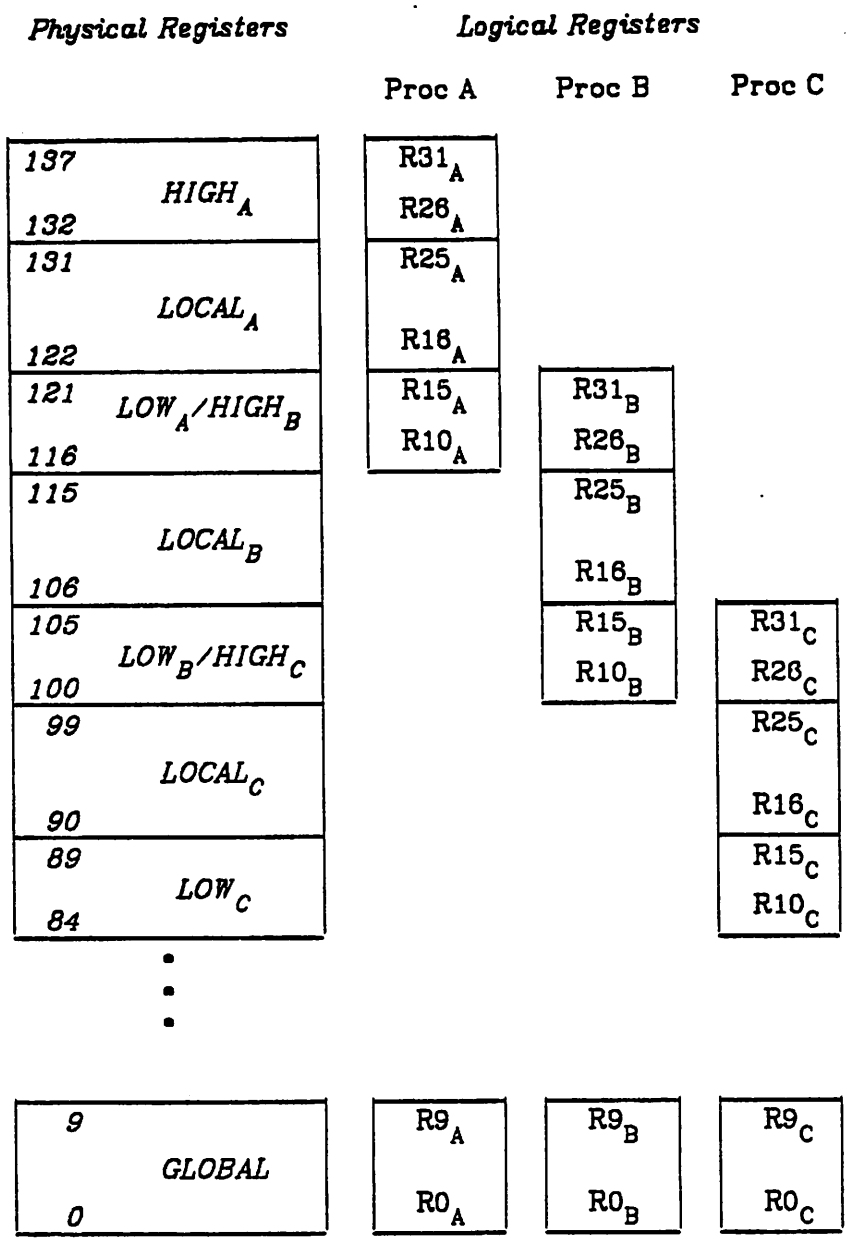

Figure 3. Usage of Three Overlapped Register Windows.

compiler determine what variables have pointers and put such variables in memory. This precludes separate compilation and slows access to these variables. 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 comparison, whether a register address points to a register in the CPU or whether it points to a register that has overflowed into 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.

This addressing technique also solves the "up-level addressing" problem. Pascal and other languages allow nested procedure declarations thereby creating a class of variables that are neither global variables nor local to a single procedure. Compilers keep track of each procedure environment using static and dynamic links or displays. Such a compiler for RISC would also associate the memory address for the window of local variables. These variables would then be accessed by using the display or dynamic chains to find the corresponding memory addresses.

Delayed Jump

The normal RISC I instruction cycle is just long enough to execute the following sequence of operations: read a register, do an ALU operation, and store the result back into a register. We increase performance by prefetching the next instruction during 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*.

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.

Table 7 illustrates the delayed branch. Machines with normal jumps would execute the sequence in Table 7(a) in the order 100, 101, 102, 105, ... To get that same effect in RISC I, we would have to insert a NOP (Table 7(b)). The sequence of instructions for RISC I is now 100, 101, 102, **103**, 106, ... In the worst case, every jump could take two instructions. The RISC I compiler, however, includes an optimizer that tries to rearrange the sequence of instructions to do the equivalent operations while making use of the instruction slot where the NOP appears. As shown in Table 7(c), the optimized RISC I sequence is 100, 101, **102**, 105, ... 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 Table 7(a).

Table 7.
Normal and Delayed Jump.

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**		

Architectural Heritage

Architects of new machines are building on the work of others, and we believe it is important to trace the genealogy of RISC I. Its earliest ancestor is the 1951 Ferranti-Manchester MADM - the first machine with index registers - which also used a register to supply zero. [15] Seymour Cray revived the idea in 1964 with the CDC-6400, and continued to use it in the CDC-7600 and the Cray 1. The delayed jump was first used in the MANIAC I, which was completed just a year after the MADM, but we adopted the idea from microprogrammed control units, where delayed jumps are the norm.

The leading proponent of reduced instruction set computers for floating point data is Cray. For the last 15 years, he has combined simple instruction sets with sophisticated pipelined implementations to create the most powerful floating point engines in the world. While Cray concentrates on impressive floating point rates at impressive costs, RISC I concentrates on improved performance at lower cost for integer programs written in HLL's.

A machine with similar goals that predates RISC is the IBM 801. This project, led by John Cocke and George Radin, began in 1975 by re-examining the relationship between instruction sets, compilers, and operating systems. They pushed the state of the art of compiler technology and created an extremely fast, reduced-instruction-set ECL minicomputer. Alas, the architecture community was left to widely varying rumors about the technical details [16] as well as the success or failure of the project. [17] Fortunately, accurate information is beginning to emerge. [18] It will be interesting to see similarities between RISC I and the 801, but differences are that RISC I uses traditional compiler technology and the 801 uses a traditional register set.

In searching the annals of computer architecture we cannot find a clear reference to overlapped register windows. To our best knowledge, no machine uses the scheme for fast, multi-port registers in the CPU. Most modern machines support procedure call by having instructions that manage a portion of main memory as a stack to pass parameters and allocate locals. Theoretically, a cache should then make such a scheme as fast as the overlapped register windows. The reasons registers are faster than caches are the difference in speed between a small memory and a large memory, the difference in speed between a deterministic access and a probabilistic access, and the difference in speed between a non-translated register access and a translated virtual memory access. Theoretically, hardware can overcome almost any obstacle, but it occasionally stumbles in implementation. The advantages of registers become apparent when we look at concrete realizations; as we shall see, procedure call/return on the VAX-11/780, using a software stack enhanced by a hardware cache, is about an order of magnitude slower than the overlapped register windows of RISC I (Table 10).

There are a few machines which share features of RISC I's overlapped register window scheme. The BBN C/70, a recent machine, allocates a new set of registers on every procedure call, but it does not overlap register sets. The most popular architecture that comes close to RISC is the Texas Instruments 990-9900 family. These machines allocate their general 'registers' in memory, so adding the contents of one register to another results in three memory accesses. A single register points to the register work space; most of the machines allow the pointer to overlap work spaces. The latest generation of this family, the TI 99000, includes on-chip main memory, but the first models appear to still have slow register access. [19] The machine that comes closest to the overlapped register windows is the Bell Labs MAC-8. The state of the NMOS technology in 1975 precluded having a rich instruction set *and* a register file on the chip;

the architects chose the rich instruction set. The main difference between the MAC-8 and TI 990 is that the Bell architects realized that overlapping the registers could improve the performance of the procedure call and provided instructions to specifically overlap the register windows in memory. It is our understanding that some C compilers used this feature. This machine was never implemented with on-chip registers, and the logical successor to this machine, the BELLMAC-32, has abandoned this approach.

VLSI IMPLEMENTATION

The transition from theoretical architecture to concrete circuits began began on January 6, 1981. Mask descriptions were completed June 22 and we received first silicon on October 23. Figure 4 is a photomicrograph of RISC I. We followed the Mead-Conway design philosophy for NMOS with lambda at 2 microns and no buried contacts. This first version, RISC I 'Gold' as it is known internally, implements the complete instruction set and 6 windows with a total of 78 registers. The only piece of the architecture that was not implemented is the mapping of registers into the memory address space.

We collected statistics on the design and layout of RISC I. [20] Table 8 compares these results to VLSI implementations of more complex architectures. The most visible impact of the reduced instruction set is the reduced control area: control is only 6 % of RISC I compared to 50 % in others. RISC I is 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. [21] 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 the masks was 14 months, less than the development periods of other machines. This was due in part to the reduced instruction set and in part to the Berkeley CAD software that was a good match for this style of VLSI design. The primary interface was Caesar, an excellent color graphics layout editor developed by Ousterhout. [22]

Table 8.
VLSI Design Metrics for Z8000, MC68000, iAPX-432, and RISC I.

	Zilog Z8000	Motorola 68000	Intel iAPX-432			RISC I
			43201	43202	43203	
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 Effort (man months)	60	100	170	170	130	15
Layout Effort (man months)	70	70	90	100	50	12

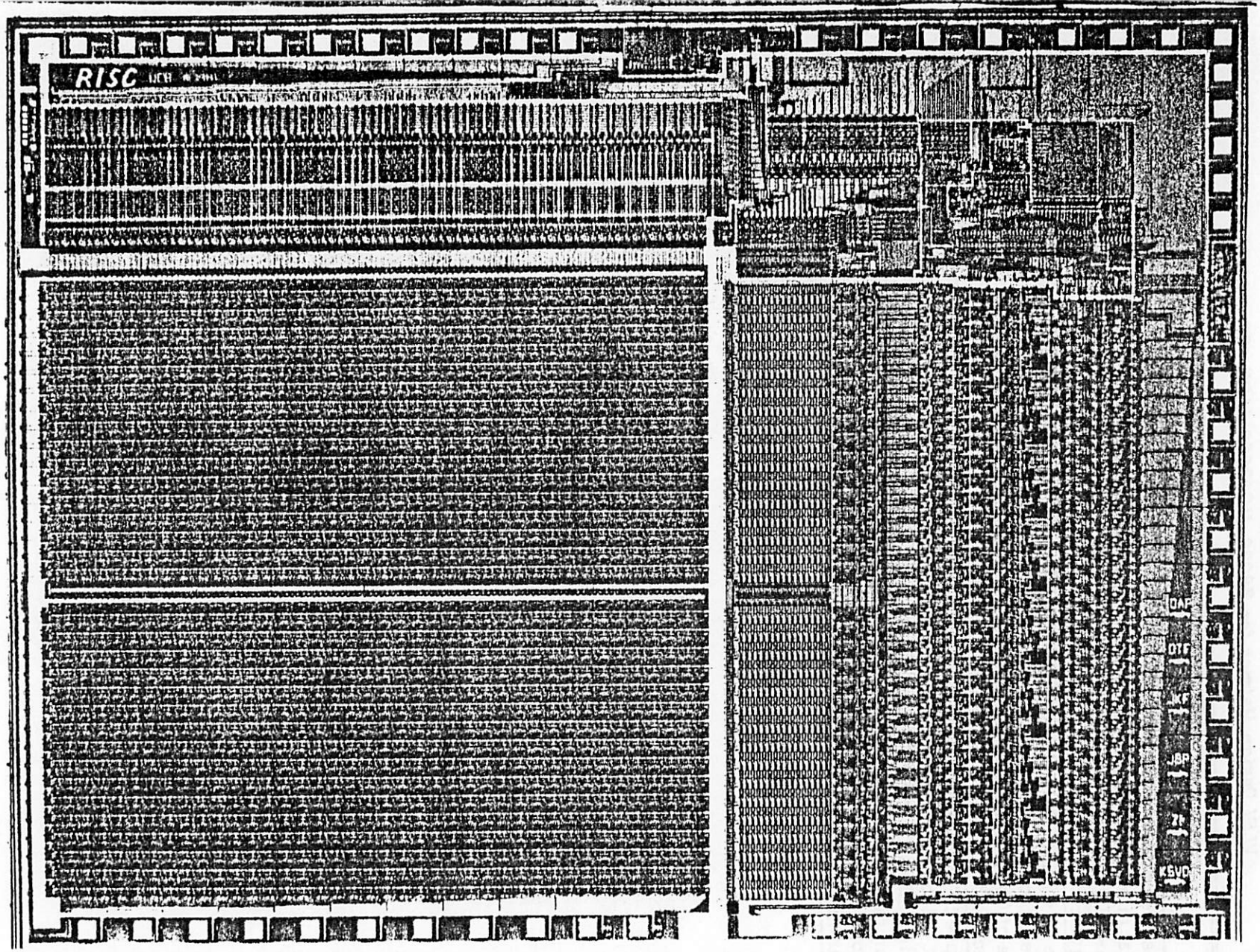

Figure 4. Photomicrograph of RISC I.

EVALUATION

This section will discuss the overall performance of RISC I and assess the contributions of the register window scheme and the delayed branch.

Register Windows.

The results of two benchmarks presented below show that the window registers are effective in reducing the cost of using procedures. "Puzzle" and "quicksort" are two recursive programs that behave quite differently. Quicksort has a large percentage of procedure calls, while puzzle has such a low density of calls that it is almost atypical for modern structured programs. Puzzle does have a large nesting depth. In both cases, the window scheme proves to be beneficial. Table 9 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 due to saving and restoring registers in the 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 of .37 words are transferred to memory per procedure invocation for the puzzle program and .07 for quicksort. Note that half of the data memory references in quicksort are the result of the call/return overhead of the VAX.

Table 9.
Memory Traffic Due to Call/Return

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

The next table compares the average "cost" of a procedure call/return pair measured in execution time, number of instructions executed and data memory accesses, of the RISC I procedure mechanism to that of three traditional machines. The data was collected by looking at the code generated by C compilers for these machines for procedure call and return statements, assuming that two parameters are passed and requiring that three registers be saved.

Table 10.
Procedure Call/Return Overhead (including parameter passing)

	Execution Time (usecs)	Instructions Executed	Data Memory accesses
VAX-11	26	5	19
PDP-11	22	19	15
68000	19	9	12
RISC I	2	6	0.2

The window 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. [23,24] Because RISC I arithmetic and logical instructions cannot access memory, one might expect an even higher fraction of the instructions to be data transfer. This is not the case. The static frequencies of RISC I instructions for nine typical C programs show that less than 20% of the instructions are loads and stores while more than 50% of the instructions are register-to-register. RISC I has successfully changed the allocation of variables from memory into registers, thus minimizing the slower off-chip memory accesses. This demonstrates that complex addressing modes are not necessary to obtain an effective machine.

Delayed Jump

The effectiveness of rearranging the code around jump instructions can be evaluated by counting the NOP instructions in a program. Static figures before optimization show that in typical C programs about 18% of the instructions are NOP's inserted after jump instructions. A simple peephole optimizer reduces this to about 8%. The optimizer does well on unconditional branches (removing about 90% of the NOP's) but not so well with conditional branches (removing only about 20% of the NOP's). Note that these are the *static* numbers, and the *dynamic* numbers can be worse depending on program structure.

This optimizer was improved to replace the 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's except for those that follow return instructions, dropping NOP's from 12% statically to 3%. The dynamic effectiveness of the delayed branch must now include the NOP's plus the instructions after conditional branches that need not be executed for a particular jump condition. The total percentages of either type of instruction are again program dependent, ranging from 4% to 22%.

Overall Performance

Prototype versions of a RISC I compiler for 'C', optimizer, linker, assembler, and simulator were developed early in the project to predict the code size and performance of RISC I. The minicomputers and microprocessors chosen for this comparison are described in Table 11.

Table 11.
Characteristics of Six Machines.

	Microprocessors NMOS VLSI			*Minicomputers* Shottky TTL MSI		
	RISC I	68000	Z8002	VAX-11/780	PDP-11/70	C/70
Year of Introduction	1981	1980	1979	1978	1975	1980
# of Basic Instructions	31	61	110	248	65	40
# of General Registers	32	15	14	13	6	8
# of Addressing Modes	2	14	12	18	12	17
Address Size(bits)	32	24	16	32	16	20
Basic Clock Frequency	7.5MHz	10MHz	6MHz	5MHz	7.5MHz	6.7MHz
Reg. to Reg. Add (usec)	0.4	0.4	0.7	0.4	0.5	?
Modify Index, Branch if Zero (branch taken)	1.2	1.0	2.2	1.4	0.8	?

We didn't have working hardware for the either the 68000 or RISC I, so we used simulators to predict performance. The cycle time for the first RISC I prototype

is expected to be 400 nsec: read and add two 32-bit registers, store the result in a register, and prefetch the next instruction. This estimate is both optimistic and pessimistic: optimistic in that it is unlikely that students can successfully build something that fast on their first try, and pessimistic because an experienced IC design team could build a much faster machine.

We chose 11 C programs for the performance comparison. The first five programs are HLL versions of the "EDN" benchamrks. [25] 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 compilers used are quite similar; the VAX, C/70, Z8002, 68000, and RISC C compilers are based on the UNIX Portable C Compiler, [26] and the one for the PDP-11 is based on the Ritchie C compiler. [27] Experiments comparing the Ritchie and Portable C Compilers for the PDP-11 have shown that the average difference in the size of generated code is within 1%. [28]

Tables 12 and 13 compare the relative performance and code size of these minicomputers and microprocessors on the eleven C programs.

Table 12.
C Benchmarks: RISC I Program Size and RISC I Size Ratio

BENCHMARK	RISC I	68000	Z8002	VAX-11/780	11/70	C/70
	bytes	Program Size Relative to RISC I				
E - string search	144	.8	.9	.7	.8	.7
F - bit test	120	1.2	1.5	1.2	1.4	1.0
H - linked list	176	.7	.8	1.2	1.7	.8
K - bit matrix	288	1.1	1.3	1.0	1.3	1.1
I - quicksort	992	.7	1.1	.9	1.1	.9
Ackermann(3,6)	144	---	2.1	.5	.6	.5
puzzle(subscript)	2736	---	.5	.5	.6	.6
puzzle(pointer)	2796	.9	.5	.5	.5	.6
recursive qsort	752	---	.8	.6	.8	.6
sed(batch editor)	17720	---	1.0	.6	.5	.5
towers Hanoi(18)	96	---	2.5	.8	1.0	.7
Average±S.D.		.9 ± .2	1.2 ± .6	.8 ± .3	.9 ± .4	.7 ± .2

Table 13.
C Benchmarks: RISC I Execution Time and RISC I Performance Ratio

BENCHMARK	RISC I msecs	68000	Z8002	VAX-11/780	11/70	C/70
		Number of Times Slower Than RISC I				
E - string search	.46	2.8	1.6	1.3	0.9	2.2
F - bit test	.06	4.8	7.2	4.8	8.2	9.2
H - linked list	.10	1.6	2.4	1.2	1.9	2.5
K - bit matrix	.43	4.0	5.2	3.0	4.0	9.3
I - quicksort	50.4	4.1	5.2	3.0	3.6	5.8
Ackermann(3,6)	3200	---	2.8	1.6	1.6	---
recursive qsort	800	---	5.9	2.3	3.2	1.3
puzzle(subscript)	4700	---	4.2	2.0	1.6	3.4
puzzle(pointer)	3200	4.2	2.3	1.3	2.0	2.1
sed(batch editor)	5100	---	4.4	1.1	1.1	2.6
towers Hanoi(18)	6800	---	4.2	1.8	2.3	1.6
Average±S.D.		3.5 ± 1.8	4.1 ± 1.6	2.1 ± 1.1	2.6 ± 1.5	4.0 ± 2.8

A surprising result is that RISC programs are at worst a factor of two larger than programs for the other machines even though size optimization was virtually ignored. To us the most important figure of merit for a new architecture is execution time. Table 13 shows that RISC I executes C programs faster than currently available microprocessors - faster even than most minicomputers.

DISCUSSION

The presentation of the RISC concept has led to many stimulating discussions. Listed below are frequently heard comments *(in italics)* followed by a short discussion of that comment.

CISC's provide better support of HLL since they include HLL primitives (CASE, CALL).

CISC architectures support HLL's by narrowing the gap between the semantics of the assembly language and the semantics of a HLL. Support can also, however, be measured as the inverse of the "costs" of using typical HLL constructs on a particular machine. If the architect provides a feature that "looks" like the HLL construct, but runs slowly, the compiler writer will omit the feature or, worse, the HLL programmer concerned with performance will avoid the construct. A recent study shows that CISC's penalize the use of HLL far more than RISC's. [29]

It is more difficult to write a compiler for a RISC than a CISC.

A recent paper by Wulf [30] 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 and addressing modes), the more cases must be considered. The compiler writer must balance the speed of the compiler with his desire to get good code. In CISC's there may not be enough time to analyze the potential usage of all available instructions. This explains Wulf's recommendation that

"There should be precisely one way to do something, or all ways should be possible."

In RISC we have taken the former approach. There are few choices; for example, if an operand is in memory, it must first be loaded into a register. Simple case analysis implies a simple compiler, even if more instructions must be generated in each case.

RISC I is tailored to C and will not work well with other HLL's.

Studies of other HLL's [23,31] indicate that the most frequently executed operations are the same simple HLL constructs found in C, for which RISC I has been optimized. Unless a HLL significantly changes the way people program, we expect to see similar results. In the case of languages that have unusual data types, such as COBOL, we need to find the simple operations that are used repeatedly in that environment and incorporate them into a RISC. Even if COBOL does not map efficiently onto the RISC I architecture, we believe the same philosophy can lead to a RISC that does.

Comparisons of RISC with the VAX are unfair in that the VAX provides a virtual 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. [32] If we were to put this chip in a system with a RISC CPU it would add another 100 ns to every memory access. Memory is referenced every 400 ns in RISC I, so such a combination would reduce RISC performance by 20%. Because 80% to 90% of memory references in RISC I are to instructions, [1] more sophisticated approaches, such as translating addresses only when crossing a page boundary, might limit performance reduction to only 5%. A final observation is that even if the addition of virtual memory doubled the cycle time of NMOS RISC I, it would still be faster than most present-day microprocessors.

The good performance is due to 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. A key point is that there would have been no room for register windows if control had not dropped from 50% to 6%. Furthermore, control is so simple in RISC that microprogramming in unnecessary; this eliminates the control loop as the limiting factor of the machine cycle, as is frequently the case in microprogrammed machines.

There is no difference between overlapped registers windows and a data cache.

A cache is ineffective if it is too small. An effective data cache would require a much larger area than our register file, especially if it must provide the same number of ports as the register file. The more complicated virtual address translation and decoding would likely stretch the basic CPU cycle time. Finally, the more complicated cache control would have extended the design phase of RISC I.

CONCLUSION

RISC I is a representative of a new style of computers that take less time to build and yet provide higher 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. 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 such a single-chip computer much sooner and with less effort than traditional architectures.

As we go to press, we are just testing the RISC I chips. Unfortunately, the polysilicon layer was processed improperly, and we believe this accounts for the fact that the chips are only partially operational. We have not yet found any circuit design errors.

This research area is by no means closed. For example, an investigation of a RISC with two ALU operations per cycle and dual port main memory has begun at Stanford, [33] and we are working on a new implementation with a denser register file and a more sophisticated timing scheme. [34] Some of the other topics to be investigated include the applicability of RISC's to other HLL's (e.g., LISP, COBOL, Ada), the effectiveness 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 project; we hope to cooperate with industry and academia in exploring RISCy architectures.

ACKNOWLEDGEMENTS

The RISC Project has been sustained by a large group of volunteers. We would like to thank all those in the Berkeley community who have helped push RISC from a concept to a chip. We would also like to give special thanks to a few.

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, supplying technical and managerial expertise. We 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. Many have participated but a few contributed significantly. Manolis Katevenis did the initial block structure, the initial timing description, and provided many important simplifications and ideas about the implementation and the architecture. Ralph Campbell wrote the initial C compiler, the optimizer, assembler, and linker. Yuval Tamir wrote a simulator, ran the benchmarks[35] and provided many suggestions in the initial design of RISC I. Gary Corcoran wrote the initial ISPS description of RISC I. Jim Peek, Korbin Van Dyke, John Foderaro, Dan Fitzpatrick, and Zvi Peshkess were the principal VLSI designers of the first RISC I chip. Michael Arnold, Dan Fitzpatrick, John Foderaro, and Howard Landman all wrote CAD tools that were crucial to the VLSI implementation of RISC I. Peter Kessler helped derive the overlapped register windows and helped with the CAD software. Jim Beck and Bob Cmelik created the VLSI testing hardware and software. Bob Sherburne is currently working with Katevenis on a more efficient VLSI implementation of RISC. Earl Cohen and Neil Soiffer collected statistics on C programs and Shafi Goldwasser collected similar statistics for Pascal.

I would also like to thank Korbin Van Dyke for his useful suggestions on improving this paper.

This research was 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. 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).
2. Strecker, W.D., "VAX-11/780: A Virtual Address Extension to the DEC PDP-11 Family," *Proc. NCC*, pp. 967-980 (June 1978).
3. Utley, B.G.etal, *IBM System/38 Technical Developments*, IBM GS80-0237 1978 .
4. Colley, S., Cox, G., Lai, K., Rattner, J., and Swanson, R., "The Object-Based Architecture of the Intel 432," *COMPCON*, (February 1981).
5. 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).
6. Patterson, D.A. and Séquin, C.H., "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers* C-29(2) pp. 108-116 (February 1980). Joint Special Issue on Microprocessors and Microcomputers.
7. Ditzel, D.R. and Patterson, D.A., "Retrospective on High-Level Language Computer Architecture," *Proc. Seventh Annual International Symposium on Computer Architecture*, pp. 97-104 (May 6-8, 1980).
8. Wulf, W., *Private Communication*. November 1980.
9. Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architecture," *Comm. ACM* 20(3) pp. 143-153 (March 1977).
10. Wichmann, B.A., "Ackermann's Function: A Study in the Efficiency of Calling Procedures," *BIT* 16(1) pp. 103-110 (1976).
11. Baskett, F., *A VLSI Pascal Machine*, Public Lecture, University of California, Berkeley, Fall 1978.
12. Sites, R.L., "How to Use 1000 Registers," *Caltech Conference on VLSI*, (January 1979).
13. Halbert, D. and Kessler, P., *Windows of Overlapping Registers*, CS292R Final Reports, June 9, 1980.
14. Morris, D. and Ibbett, R.N., *The MU-5 Computer System*, Springer-Verlag, 1979.
15. Williams, F.C. and Kilburn, T., "The University of Machester computing machine," *Inaugural Conference of the Machester University Computer*, pp. 5-11 (July 1951).
16. Anonymous,, "Altering Computer Architecture is Way to Raise Throughput, Suggests IBM Researchers," *Electronics*, pp. 30-31. (December 23, 1976).
17. Anonymous,, "IBM Mini a Radical Departure," *Datamation*, pp. 53-55 (October 1979).
18. Radin, G., "The 801 Minicomputer," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, (March 1-3, 1982).
19. Orlando, R.V. and Anderson, T.L., "An Overview of the 9900 Microprocessor Family," *IEEE MICRO* 1(3) pp. 38-42 (August 1981).
20. 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* II(4) pp. 14-20 (Fourth Quarter (October, 1981)).

21. 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).

22. Ousterhout, J., "Caesar: An Interactive Editor for VLSI Circuits," *VLSI Design* II(4) pp. 34-38 (November, 1981).

23. Alexander, W.C. and Wortman, D.B., "Static and Dynamic characteristics of XPL Programs," *Computer* 8 (11) pp. 41-46 (November 1975).

24. Shustek, L., *Analysis and Performance of Computer Instruction Sets*, Ph.D. Thesis, Stanford University January 1978.

25. Grappel, R.G. and Hemmengway, J.E., "A Tale of Four Microprocessors: Benchmarks Quantify Performance," *Electronic Design News*, pp. 179-265 (April 1, 1981).

26. Johnson, S.C., "A Portable Compiler: Theory and Practice," *Proc. Fifth Annual ACM Symposium of Programming Languages*, pp. 97-104 (January 1978).

27. Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1978).

28. Johnson, S.C., *Private Communication*. January 1981.

29. 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).

30. Wulf, W.A., "Compilers and Computer Architecture," *Computer* 14(7) pp. 41-48 (July 1981).

31. Ditzel, D.R., "Program Measurements on a High-Level Language Computer," *Computer* 13(8) pp. 62-72 (August 1980).

32. Lavi, Y., Kaminer, A., Manachem, A., and Bash, S., "16-bit microprocessor enters virtual memory domain," *Electronics*, pp. 123-129 (April 24, 1980).

33. Hennessy, J., Jouppi, N., Baskett, F., Strong, A., Gross, T., Rowen, C., and Gill, J., "The MIPS Machine," *Proc. Compcon*, (February 1982).

34. Sherburne, R.W., Katevenis, M.G.H., Patterson, D.A., and Séquin, C.H., "Datapath Design for RISC," *Proc. Conference on Advanced Research in VLSI*, pp. 53-62 (January 25-27, 1982).

35. Tamir, Y., "Simulation and Performance Evaluation of the RISC Architecture," Electronics Research Laboratory Memorandom No. UCB/ERL M81/17, University of California, Berkeley, Cal. (March 1981).