DESIGN AND IMPLEMENTATION OF RISC I

C.H. Séquin and D.A. Patterson

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

ABSTRACT

The Reduced Instruction Set Computer (RISC) is an architecture particularly well suited for implementation as a single-chip VLSI computer. It demonstrates that by a judicious choice of a small set of instructions and the design of a corresponding micro-architecture, one can obtain a machine with high throughput. The limited number of instructions and addressing modes leads to a small control section and to a short machine cycle time. Such a machine also requires a much smaller layout effort and thus leads to a shorter design cycle.

Such a RISC architecture has been implemented at U.C.Berkeley as part of a four quarter sequence of graduate courses in which students propose and evaluate architectural ideas, design LSI components, integrate these components into a VLSI chip, and finally test the actual chip. The CAD and testing environment in which this chip was created is also described.

Proc. Advanced Course on VLSI Architecture
University of Bristol, England, July 19-30, 1982.

DESIGN AND IMPLEMENTATION OF RISC I

C.H. Séquin and D.A. Patterson

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

1. INTRODUCTION

Advances in VLSI technology make it possible to realize the minicomputers of yesteryear on a single chip of silicon. However, this new implementation presents constraints that are quite different from those of main-frame technology. For a long time to come, a single chip of silicon represents a rather limited resource in terms of the number of transistors it can accommodate and the amount of power dissipation it can handle.[Patt80b] Furthermore, because of the scaling laws of MOS technology, the active devices on the chip will get ever smaller and faster, so that the wiring between the devices will soon become the dominant problem. Intra-chip communication must thus be carefully addressed; random logic and long-distance connections need to be minimized. A clean floor plan relying on regular arrays with high device density is very desirable since it also simplifies the layout task.

To build an effective single-chip computer, one must thus not simply map the architecture of a successful minicomputer onto the surface of a silicon crystal. The architecture must first be redesigned with the above constraints in mind. Because of the relative delay and power penalty of sending signals from one chip to another, systems partitioning has to be addressed very carefully, and the right combination of elements must be grouped together on a single chip. The limited number of transistors need to be allocated judiciously to the processor, main memory, communication ports, and other desired functions.

In this context, we found that a judicious restriction to a small set of often used instructions, combined with an architecture tailored to fast execution of all the instructions in this set, can result in a machine of surprisingly high effective throughput. Such a *Reduced Instruction Set Computer (RISC)*[Patt81, Patt82c] can be realized with a small control section and a comparatively short machine cycle. In addition to being a more suitable match for VLSI, this approach dramatically reduces the long design times and the high incidence of architectural design flaws and inconsistencies, both typically associated with the first prototypes of computers of traditional design.

Students taking part in a multi-term course sequence designed a complete 32-bit NMOS microprocessor called RISC I.[Fitz81] This first design, previously also referred to as the "*Gold*" chip, was finished in June 1981 and in the meantime has been implemented by MOSIS (DARPA's MOS Implementation Service at the University of Southern California's Information Sciences Institute)[Cohe82] and tested and evaluated.[Fode82] In parallel, Katevenis and Sherburne started from

the basic organization of RISC I and introduced a more compact register file, which required, however, a more sophisticated timing scheme. This more ambitious design, called RISC II or the "*Blue*" chip, has almost twice the local memory capacity, but still fits onto a smaller chip than RISC I. It will be submitted for fabrication in Fall 1982.

Similar experiments are being carried out in other places. Particularly noteworthy are IBM's 801 project initiated by John Cocke in the mid 1970's and led by G. Radin[Radi82] as well as the MIPS project at Stanford.[Henn81, Henn82]

2. DESIGN GOALS FOR RISC I

The RISC project started with an intensive six-month study phase during which the basic concept was evaluated. RISC I was designed with particular attention to the needs of high-level language programming. The selection of languages for consideration in RISC I was influenced by our environment; we chose *C* and *Pascal* since there is a large user community and considerable local expertise. Given the limited number of transistors that can be integrated at present onto a single chip, most of the pieces of a RISC I system are in software, with hardware support for only the most time-consuming events. To the user it should not matter whether a high-level language computer system is implemented mostly by hardware or mostly by software, provided the system is efficient and hides any lower levels from the programmer.[Patt80a] This approach requires an efficient compiler and HLL debugging tools that give all error messages in the context of the source code. Given this framework, the role of the architect is to build a cost-effective system by deciding which pieces of the system should be in hardware and which in software.

Because of the bandwidth bottleneck at the chip periphery, the emphasis in a VLSI chip must be on self-contained action. Most of the RISC I instructions are thus "register-to-register" and take place entirely inside the chip. Data memory access is restricted to the LOAD and STORE instructions. The instructions are kept simple so that they can be executed in a single, short machine cycle; and they are each one word (32-bits) long to avoid the hardware complexity associated with variable-length instructions. Less frequent operations are implemented with instruction sequences or subroutines.[Patt81, Patt82c]

The relative dynamic frequencies of high-level language statements show which constructs are used most often, but they are a poor measure of the actual effort of the computer devoted to particular classes of statements. To determine which statements use the most time in the execution of typical programs, one must look at the code produced by typical versions of each of these statements and multiply the frequency of occurrence of each statement with the corresponding number of machine instructions or memory references. This gives a better estimate of the relative "cost" of each statement type (Table 1).

The data in this table indicate that the procedure call/return is the most time-consuming operation in typical high-level language programs. RISC I programs potentially have an even larger number of calls since some of the complex instructions found in traditional architectures are implemented as subroutines. Thus the procedure call must be as fast as possible. Other statistics taken on the occurrence of various operands show the importance of local

Table 1.
Relative Frequency of HLL Statements.
(ordered by memory references)

statements HLL	HLL (# occurrence) P	C	WEIGHTED (# instr.) P	C	WEIGHTED (# mem. ref.) 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

variables and constants: more than 80 % of all dynamic scalar references are to local variables. RISC I & II support these constructs with especially large register files. Arrays or structures, on the other hand, are typically shared global variables and are kept in main memory.

3. OVERLAPPING REGISTER BANKS

The use of procedures involves two groups of time-consuming operations: saving or restoring registers on each call or return, and passing parameters and results between procedures. This overhead can be reduced if the processor is equipped with multiple banks of registers.[Site79, Bask78] The frequency of local scalar variables justifies architectural support by placing locals in registers.

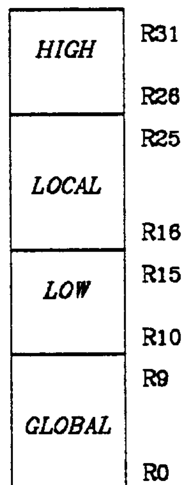

Figure 1. *Naming within one Virtual RISC II Register Window.*

In RISC I the chip area saved by the simplicity of the control circuitry was devoted to an extra large set of 32-bit registers. The processor allocates a new

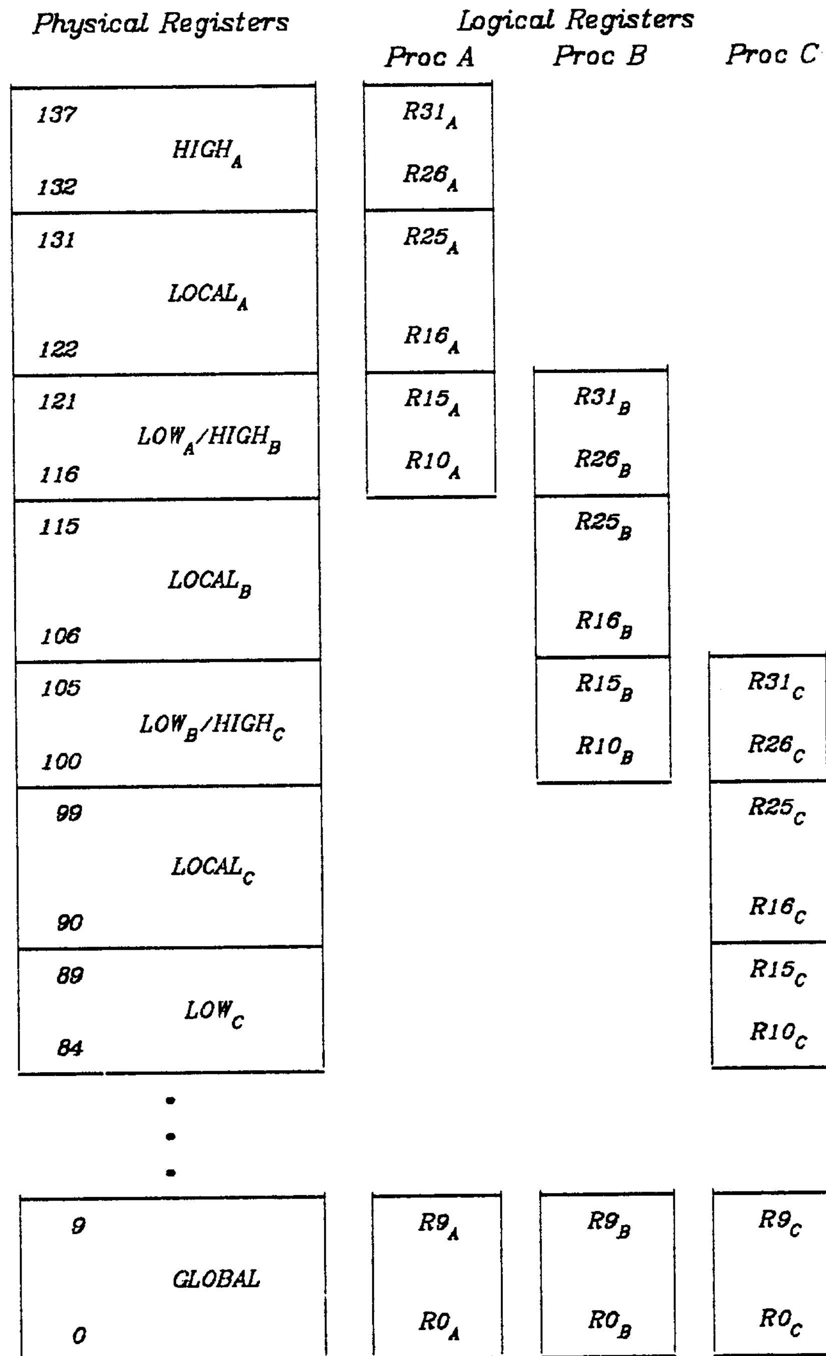

Figure 2. *Usage of Overlapped Register Windows (RISC II).*

bank of registers for each procedure call by simply changing a hardware pointer, thus avoiding the overhead of saving registers in memory. The return instruction resets the register bank pointer to the previous value, which restores the old set of register values. There are also ten *global* registers, thus giving every procedure access to a total of 32 registers as shown in Figure 1.

Registers 26 through 31 (*HIGH*) contain parameters passed from "above" the current procedure, i.e., the calling procedure. Registers 16 through 25 (*LOCAL*) are used for the local scalar storage. Registers 10 through 15 (*LOW*) are used for local storage and parameters passed to the procedure "below" the current procedure, i.e., the called procedure.

In addition, "neighboring" register banks used by *calling* and *called* procedures physically overlap, so that parameters may be passed to a procedure without moving any data. On each procedure call a new set of registers, named 10-31, is allocated. However, the LOW registers of the "caller" become the HIGH registers of the "callee" since they are physically the same. Thus, without moving information, parameters in registers 10-15 appear in registers 25-31 in the called frame. Figure 2 illustrates this approach for the case where procedure A calls procedure B which calls procedure C. Overall, this scheme dramatically reduces the number of accesses to data memory.

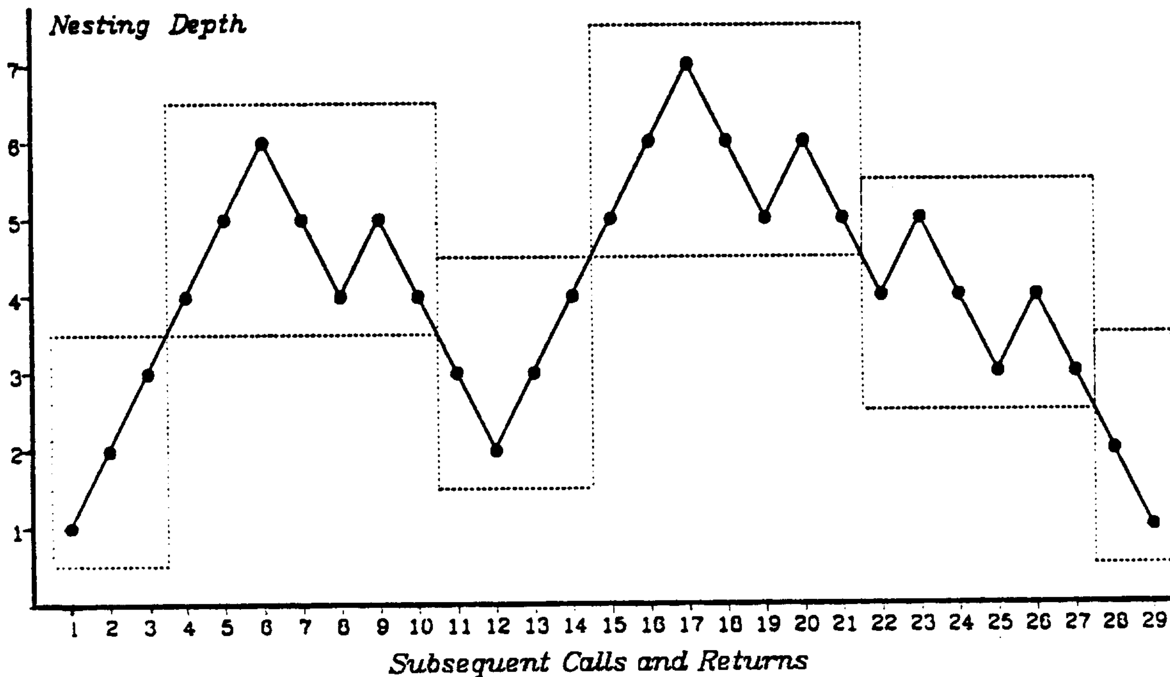

Figure 3. *Procedure Nesting Depth and Optimal Usage of a Register File with Three Banks.*

In many programs the nesting depth of procedure calls will exceed the number of register windows provided on the processor. A mechanism must be provided to free up some of the register banks by moving their contents to main memory. We have studied the sequences of procedure call/returns in several programs.[Tami82] A typical behavior of the resulting *Procedure Nesting Depth* for a small recursive program is shown in Figure 3. For this illustration it was assumed that the register file contains three banks. Each dashed frame in Figure 3 indicates the range in the nesting depth that can be handled without overflow/underflow. When the nesting depth goes outside the logical sets of registers currently contained in the physical register file, a hardware trap will

start an interrupt handler which moves a number of registers to/from main memory. A separate register overflow stack is kept in a dedicated area in memory. Overflows and underflows will also adjust a pointer to the top of this stack. The effectiveness of this procedure call/return support depends on the rarity of the occurrences of such overflows/underflows. Since the register file will always contain the top few procedure activation records, the overflow/underflow frequency is based on the local variations in the depth of the stack rather than on the absolute depth. Our studies indicate that with eight register banks overflows/underflows will occur in less than 1% of all calls/returns.

In order to make the variables in the registers also accessible via pointers, they need to be given addresses. For this purpose, all registers in the RISC architecture are also mapped into the regular memory address space. A single address comparison and an 8-input AND gate can determine whether an address points to a register bank currently on chip or to a location in memory. 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. A RISC compiler could use the memory addresses of the windows for this purpose. However, this scheme has not actually been implemented in the first versions of the RISC I & RISC II chips.

4. RISC I ARCHITECTURE

The main goal with RISC I was to obtain as much performance for as little complexity as possible. Most modern microprocessors have far more complexity built into the chip than can be warranted by the resulting performance. A Turing machine, on the other hand, while having conceptually the minimal complexity required of a general-purpose computer, obviously has an unacceptably low performance. In RISC I the complexity and the performance benefits of all features were carefully evaluated. The added complexity of the multiple overlapped register banks was introduced since it simplifies address calculations and reduces the traffic between the processor and the (off-chip) memory – a major bottleneck in most computer systems.

In the final instruction set there are no big surprises. It has 31 instructions in a few very similar formats, all 32 bits long. (Actually, a few more meaningful instructions fell out almost for free near the end of the design of RISC I, but they were not part of the original design and the compiler and assembler do not know about them.) RISC I & II support 32-bit addresses and 8-, 16-, and 32-bit data. As shown in Table 2, the instructions can be grouped into four categories: arithmetic-logical, memory access, branch and miscellaneous. All the arithmetic, logical, and shift instructions operate between registers. 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. This execution cycle is overlapped with the prefetch and decoding of the next instruction.

Load and store instructions move data between registers and memory. These instructions use two CPU cycles. We decided to make an exception to our

Table 2.
Assembly Language Definition for RISC I

Instr.	*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	*(Rx)S2,Rm*	*M[Rx+S2] ← Rm*	*store long*
STS	*(Rx)S2,Rm*	*M[Rx+S2] ← Rm*	*store short*
STB	*(Rx)S2,Rm*	*M[Rx+S2] ← Rm*	*store byte*
JMP	*CON,S2(Rx)*	*pc ← Rx+S2*	*conditional jump*
JMPR	*CON,Y*	*pc ← pc + Y*	*conditional relative*
CALL	*S2(Rx),Rd*	*CWP--; Rd ← pc, next*	*call reg.-indexed*
		pc ← Rx+S2	*and change window*
CALLR	*Y,Rd*	*CWP--; Rd ← pc, next*	*call relative*
		pc ← pc + Y	*and change window*
RET	*(Rx)S2*	*pc ← Rx+S2, next CWP++*	*return, change window*
RETINT	*(Rx)S2*	*pc ← Rx+S2; next CWP++*	*also enable interrupts*
CALLINT	*Rd*	*CWP--; Rd ← last pc*	*also disable interrupts*
LDHI	*Y,Rd*	*Rd<31:13>←Y; Rd<12:0>←0*	*load immediate high*
GTLPC	*Rd*	*Rd ← last pc*	*to restart delayed jump*
GETPSW	*Rd*	*Rd ← PSW*	*read status word*
PUTPSW	*Rm*	*PSW ← Rm*	*set status word*

original constraint of single cycle execution, rather than to extend the general cycle to permit a complete memory access in a single cycle. There are eight variations of memory access instructions to accommodate sign-extended or zero-filled 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 by using register 0 which contains a hard-wired zero.

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. The innovative features of RISC I associated with the call and return instructions have already been discussed in the previous section.

Table 3.
Basic Instruction Format for RISC I

<table>
<tr><td>OPCODE<7></td><td>SCC<1></td><td>DEST<5></td><td>SORC1<5></td><td>IMF<1></td><td>SORC2<5></td></tr>
<tr><td>OPCODE<7></td><td>SCC<1></td><td>DEST<5></td><td>SORC1<5></td><td>IMF<1></td><td>IMM.OPRD.<13></td></tr>
<tr><td>OPCODE<7></td><td>SCC<1></td><td>DEST<5></td><td colspan="3">IMMEDIATE OPERAND <19></td></tr>
</table>

Table 3 shows the 32-bit format used by register-to-register instructions, memory access instructions, and branch 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 SORC1 and SORC2. If IMF=0, the low order five bits of SORC2 specify a register; if IMF=1, the second operand is 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 whether the condition codes will be set. Memory access instructions use SORC1 to specify the index register and SORC2 to specify the offset; data is exchanged with the register specified by DEST. One other format, which combines the last three fields to form a 19-bit PC-relative address, is used primarily by the branch instructions.

Another very worthwhile complication resulting in substantial performance gain is overlapping instruction fetch and execution. Difficulties arise with branches in the control flow. If the wrong instruction has been prefetched, the two-stage pipeline must be flushed. Several high-end machines have elaborate techniques to prefetch the appropriate instruction after the branch,[Morr79] 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 permits RISC I to always 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. The RISC I compiler includes an optimizer[Camp81] that tries to rearrange the sequence of instructions to do something useful in the instruction after the jump; if that is not possible, a *NOP* (*ADD* 0,0,0) is inserted, and the jump will thus take effectively two instructions. A simple optimization that looks for a suitable instruction in the code section before the branch point can remove about 90% of the *NOP*s after unconditional jumps but can remove only about 20% of the *NOP*s associated with conditional branches. Better results can be obtained if the instruction at the ***target*** of the jump is also considered. This technique can be applied to conditional branches if the target instruction modifies temporary resources; e.g. an instruction that only modifies the condition codes. In a benchmark run of quicksort this technique removed all *NOP*s except for those that follow return instructions. In this case the number of "useless" instructions was reduced to less than ten percent of the original number. This delayed-jump mechanism and optimization is completely hidden from the user of RISC I, who programs directly in high-level languages.

5. MICRO-ARCHITECTURE

The simplicity and regularity of RISC I & 2 permits most instruction executions to follow the same basic pattern: (1) read two registers, (2) perform an operation on them, and (3) store the result back into a register. Jump, call, and return instructions add a register (possibly PC) and an offset and store the result into the appropriate PC latch. The load and store instructions violate the original constraints: in order to allow enough time for access of the main memory, they add the index register and immediate offset during the first cycle, and perform the memory access during an additional cycle. The micro-architectures of the two implementations are determined by these characteristics.

The CPU can be subdivided naturally into the following functional blocks: the register-file, the ALU, the shifter, a set of program counter (PC) registers, the data I/O latches, the program status word (PSW) register, and control, which contains the instruction register, instruction decoder, and clock-gating circuits. Since two operands are required simultaneously, the register file needs at least two independent busses and a two-port cell design. For speed, the registers are read from dynamically precharged bit lines. This requires the following basic timing sequence: (1) register read, (2) arithmetic / logic / shift operations, (3) register write, and (4) bus precharge for the next read. The cycle time is determined by this sequence of operations. For the price of a third bus, (3) and (4) can be overlapped and phase (4) can be eliminated: while the result is written back into the register file by this extra bus, the two read busses are precharged for the following read phase. This 3-phase scheme has been adopted in the RISC I processor. The basic organization with two read-only busses (A,B) and one write-only bus (C) is shown in Figure 4.

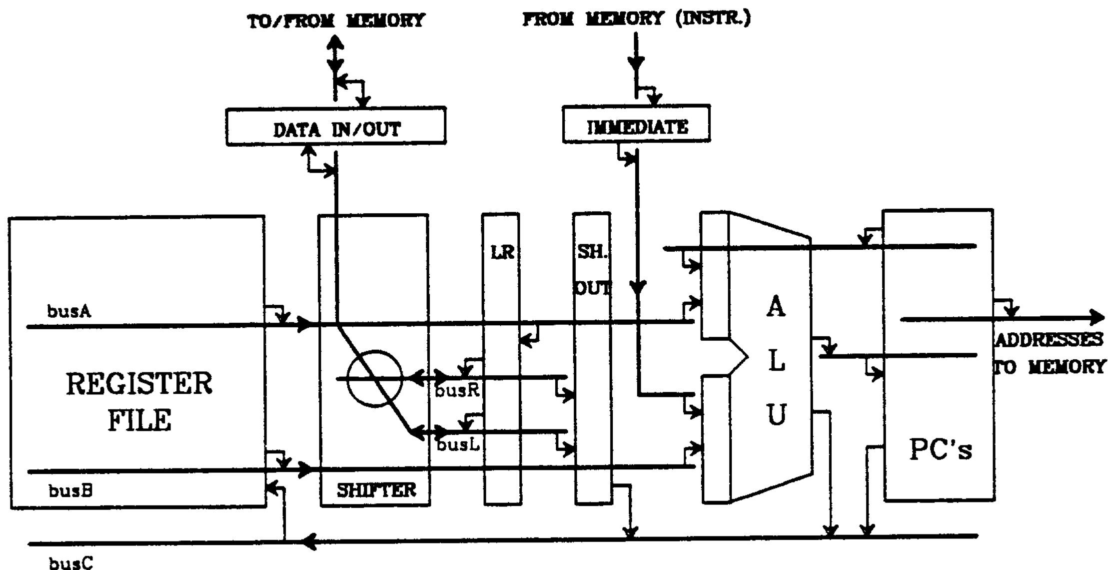

Figure 4. *The Data-Path of the RISC I Chip.*

During the evolution of the RISC I design, it became apparent that a three-bus register cell incurred a significant area penalty. Since a large fraction of the chip area is devoted to the register file, more attention was focused on the design of a smaller bit cell. The classic six-transistor static RAM cell was chosen for its compactness in the second, more ambitious RISC II chip design[Sher82]. Reading is accomplished by selectively discharging one of the two precharged bit line busses, one of which carries data in complemented form. Contrary to commercially available static RAMs, no sense amplifiers are used. This yields a speed penalty, since the bit cell must discharge a high capacitance bus, but in its place it provides a two-port reading capability. Writing is accomplished by putting both the data and its complement onto the two busses, as for a typical static RAM. The RISC II design (Figure 5) was based on this two-bus, two-port register cell.

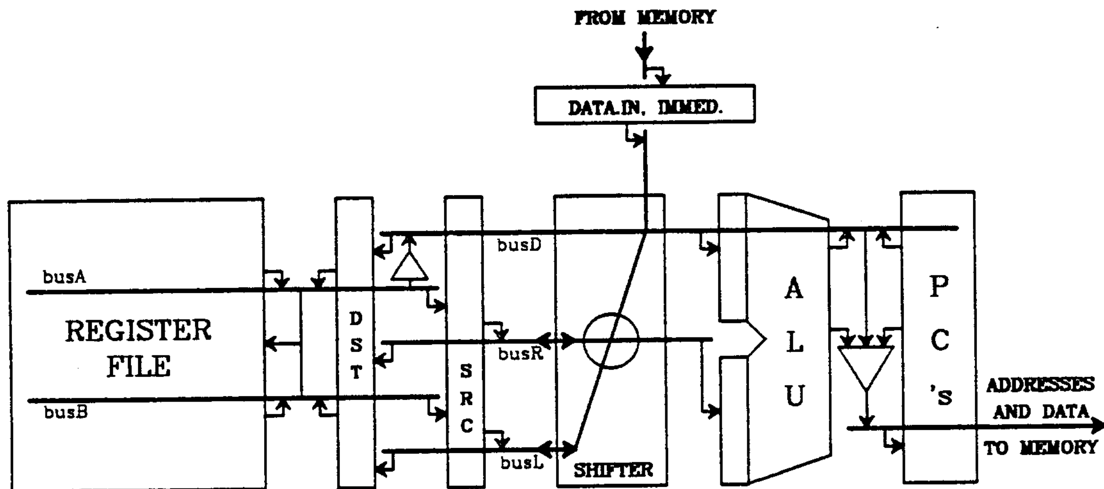

Figure 5. *The Data-Path of the RISC II Chip.*

The smaller register cell leads to a considerable reduction in chip size. There is also performance improvement due to the shorter RC delay in the poly-silicon control lines running across the data path. Further improvements were made by allowing register writing to occur in parallel with the execution of the following instruction. The result of an operation is kept in a temporary latch and is only written into the register file during the subsequent arithmetic/ logic/ shift operation phase. If the result is needed immediately in the next instruction, a register-file bypass transmits this value directly to the ALU/shifter ("internal forwarding"). In effect, each instruction now stretches over three (shorter) machine cycles: (I) Instruction fetch and decode; (II) register read, operate, and temporary latching of result; (III) write result back into the register file. In the RISC II design, these three operations are over-lapped so that a new instruction begins every machine cycle (except for *LOAD* and *STORE* instructions) as in RISC I.

Besides these changes in implementation, RISC II also incorporates an important architectural change: It was made compatible with instruction caches equipped with an "Instruction-Format Expander". As was mentioned above, an important part of RISC I's simplicity is due to the constant-length instruction-format of 32 bits. However, this approach is rather wasteful of code space. Studies by Garrison and VanDyke showed that the introduction of one additional instruction format of 16-bit length could lead to savings of 30 % in overall code size.[Patt82a] These short instructions utilize some of the previously unused opcodes, and their effects are each equivalent to the original 32-bit instructions. The RISC II CPU offers to the computer-system designer the option of improving code density for the price of an "Instruction-Format Expander," i.e. a circuit placed in the instruction-fetch path that recognizes all short instructions and translates ("expands") them into their 32-bit equivalent. Such an expander may conveniently be placed in an instruction cache. An instruction cache with a "Predictive-Program-Counter" scheme has just been designed at U.C. Berkeley, and an expander will soon be added to it.[Patt82a]

The RISC II CPU always receives 32-bit instructions, either directly from memory, or through the expander. However, in the latter case the program-counter must be incremented sometimes by 2 and sometimes by 4 in order to follow the real memory addresses of subsequent instructions and to generate correct PC-relative addresses. In the RISC II CPU there are two incrementers for the PC, one that computes PC+2, and one that computes PC+4. At the moment when a new instruction comes into the CPU, a bit that tells its original length comes in along with it. This bit selects either PC+2 or PC+4 to be sent out immediately, to start the next instruction-fetch right away.

6. THE DESIGN ENVIRONMENT

RISC I was designed with the use of a set of simple tools that performed graphic editing, check-plotting, design rule checking, layout rule checking, architectural simulation, and switch-level simulation.[Fitz81] These tools work with the CIF 2.0 geometry description[Spro80] of the chip. They all run in our UNIX environment on a VAX 11/780 and are thus readily available to the designers. This reduces the psychological overhead that often stands in the way of the usage of such tools. While many of the individual tools are still rather rudimentary, the collection of all of them together and their integration and ready availability add up to a good design environment. This environment was further enhanced by other UNIX utilities that permit effective cooperation in multi-designer teams. One such program was the *Source Code Control System (SCCS)*[Allm81] which keeps track of all incremental changes to a program or specification file, and which prevents different people from editing the same file simultaneously. In addition, the extensive use of electronic mail between the designers helped to overcome difficulties from incompatible schedules and different work habits. A special electronic bulletin board was also introduced for the RISC project to keep everybody informed about the latest developments.

The regular parts of the chip, i.e. the datapath and the register file, were generated directly at the geometry level from hand-sketched stick diagrams of the iteratively used cells. For the creation of the layout geometry we used the very effective graphics editor *Caesar*, which was developed by John

Ousterhout[Oust81] in very close interaction with the users of this tool. Both chips were designed using only "Manhattan" features (all edges parallel to x or y axes) and the simple and scalable λ-based design rules.[Lyon81] This allowed us to simplify several of the CAD tools used and thus to increase their efficiency. The layout geometry was checked with the design rule checking program *DRC* produced by C. Baker at M.I.T.[Bake80a, Bake80b] Since this program did not catch certain types of errors, e.g. too little extension of the poly-silicon gate beyond the thin-oxide channel, we counted on visual inspection to eliminate such errors.

For the control section, direct layout from the architectural description proved inadequate, and a tool was created to tie the high-level architectural description to the bit-level implementation on the chip. A multi-level description language and the corresponding simulator, *SLANG*, were developed by J. Foderaro and later refined by K. Van Dyke.[Fode81, VanD82] *SLANG* allowed the description of the whole chip at mixed levels. The random logic circuitry of the control section was described at the logic gate level. The various PLA's and decoders were described at the symbolic or boolean equation level. The data-path itself was described at the register transfer level, treating 32-bit wide data or address vectors as single entities. It was assumed that the corresponding functional blocks, such as the adder and the shifter, were implemented correctly and checked previously. The whole chip thus resulted in a *SLANG* description with less than 300 nodes, -- a number that could readily be managed by the designers. The *SLANG* description was debugged by running a dozen small diagnostic programs through the *SLANG* simulator and comparing the effect of each instruction with the ISP description[Bell70] of the architecture.

After the chip layout was completed, a detailed circuit description was obtained by running a circuit extraction program over the mask description in CIF2.0 format. A new version of a circuit extraction program, *MEXTRA*, limited to manhattan geometry and designed for high efficiency was created by D. Fitzpatrick.[Fitz82] Through the use of a clever naming convention, this extraction program already catches several types of wiring errors in the layout such as shorts between global signals (clocks) and the power supply lines. Extraction of the 44,500-transistor RISC I chip takes less than one CPU hour on the VAX 11/780.

This circuit description can then be used with circuit-level simulators to verify proper operation at the lowest level. We used *ESIM*, a switch-level simulator created by C. Terman at M.I.T[Bake80a, Term82] to test the functionality of the whole chip. For that purpose SLANG and ESIM were coupled to run in lock-step through the same diagnostic programs that had been used previously to verify the SLANG description. SLANG compared its own results with the results of the detailed switch-level simulation and reported any discrepancies. This coupled, multi-level simulation found dozens of errors that would have prevented RISC I from working.

Overall, RISC I provided a very useful forcing function for the improvement of our design environment. Many tools, when first tried on the RISC I chip, broke because of its size, or took much too long to digest this circuit with 44,500 transistors. Often the designers of the RISC team themselves sat down and improved or rewrote the tools they absolutely needed to get the job done. Since

their emphasis was not on research into CAD tools per se, the tools may be less sophisticated and less general than tools available else-where. However, the most important aspect for our work on RISC I & II was the fact that the tools were efficient, simple to use, readily available, and thus added up to a very effective design environment.

With layout rules using a λ of 2 μm, the RISC I chip is about 8 by 10 mm. The length of the chip is dominated by the data path and the register file. Only 78 instead of the desired 138 registers could be fit into the permissible length of 10 mm that was given by the size of the cavity of the selected package. These 78 registers were split into 6 windows of 14 registers each, overlapping by 4, plus 18 global registers.

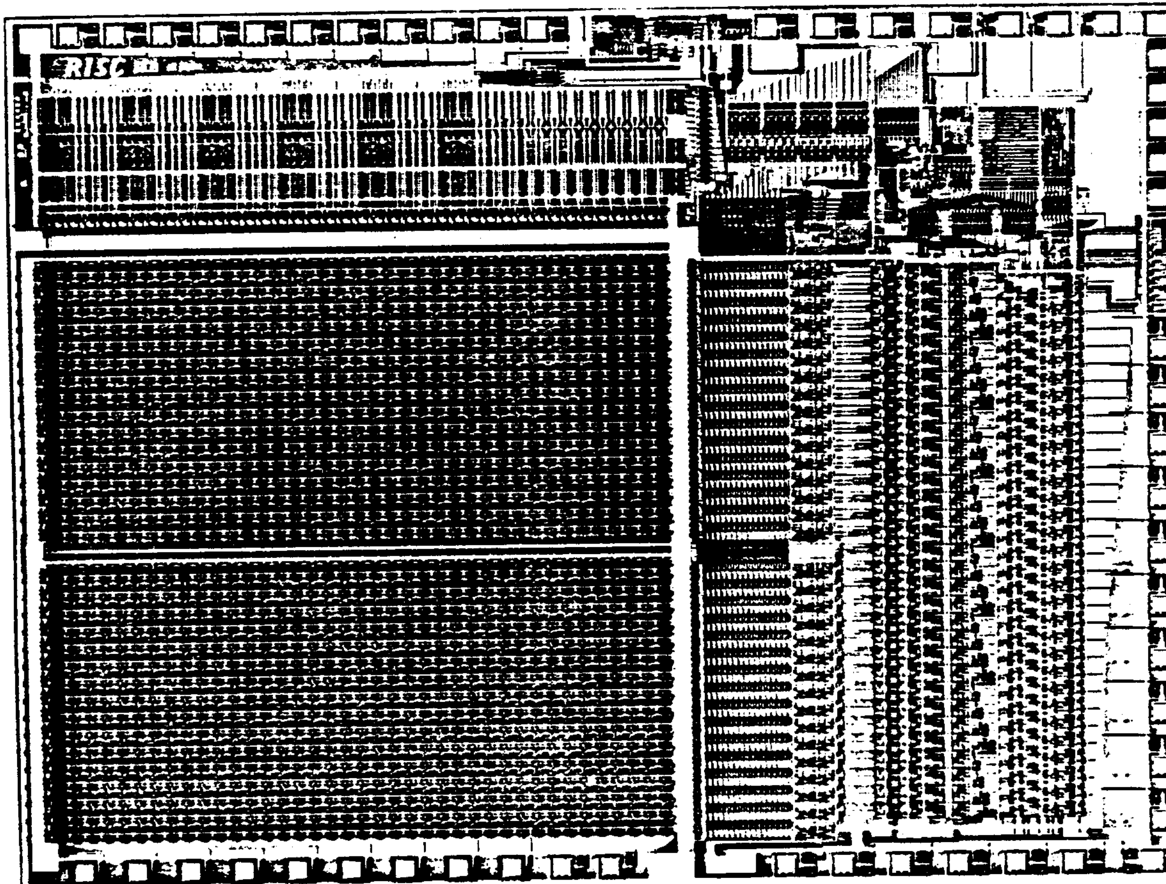

Figure 6. *Photomicrograph of the RISC I Chip.*

Control occupied only 6 % of the total chip area as compared to the more than 50 % typical for present-day commercial microprocessors.[Fitz81] It fit nicely into the upper right-hand corner left between the register file decoder and the ALU section (see Fig. 6). While the actual instruction decoding PLA turned out to be much smaller than expected, the wiring around this PLA got rather large and very tight as the chip neared completion. About half the total layout effort was associated with this small, tightly packed control section. It became evident that an interactive routing and compaction tool, integrated with the graphics layout editor, would be highly desirable for the design of the less regular sections of a chip.

Overall the RISC I processor is very regular. If we use Lattin's regularity factor,[Latt81] defined as the total number of transistors on the chip divided by the number of individually drawn devices, RISC I ends up with a regularity of 22, which is 2 to 5 times higher than the value for other microprocessors.

More than a month was spent in functional simulation and debugging of the layout after the chip was first "completed". The chip was only submitted for fabrication after functional simulations had been run flawlessly on all instructions and on several small benchmark programs.

All knowledge of the chip was kept on-line in program-understandable form that superseded any written documentation.[Fode82] Since the high-level SLANG description was written before the layout was completed, we could write special purpose programs to search this description to quickly find all the nodes that should be connected to a particular signal. Using a notebook with logic diagrams on paper as the official documentation would have been quite inadequate in our environment. The help of the computer was absolutely essential in the presence of several designers making changes in parallel. In this manner, up-to-date documentation was kept electronically in a "centralized" place, and the computer could be used to compare and correlate the various representations of the emerging design.

7. IMPLEMENTATION

The layout of the RISC I chip began on January 6, 1981. Its layout and functional simulation were completed on June 22, 1981, and the CIF description of the chip was sent to the MOSIS Implementation Service at the Information Sciences Institute associated with the University of Southern California.[Cohe82] The description was also sent to A. Bell and L. Conway at Xerox Palo Alto Research Center who had offered to include our chip on one of their own multi-project chip fabrication runs.

The RISC II design progressed at a much slower rate. It did not have the deadlines associated with the RISC I class project, but was primarily the test vehicle in the study of various trade-offs in VLSI single-chip architecture for two PhD theses. Various crucial elements of RISC I obtained significantly more attention, in particular the register file. Going to a two-bus design led to a much smaller register cell which then permitted the full complement of 138 registers to fit into the allowable chip length of 10 mm. It incorporated an additional stage of pipelining and a more sophisticated timing scheme to permit the usage of a register file with only two busses. The width of the data-path has also been reduced and the height of this chip is less than 6 mm. Control takes about the same absolute amount of space as in the first chip, but the layout was done in a more structured and less cramped manner.

The fabrication of RISC I became a challenging test case for some of the newly established implementation services.[Fode82] A sequence of human errors, bad luck, and administrative complications delayed successful implementation of this chip by many months. Four separate restarts were required to obtain working chips from the design submitted in June 1981. Wafers with good processing finally arrived in Berkeley in May 1982 from two different manufacturers. Both sets contained working chips.

In November 1981, M. Arnold finished a new design rule checker, LYRA[Arno82] This more sophisticated, corner-based checking program found some non-fatal overlap errors that had slipped through DRC and visual inspection. A corrected geometry was included on a new multi-project mask set used in January 1982. We obtained good wafers from this run in June 1982. Overall, significant progress was made during that year in the realization of an implementation service that is separated from the design and layout activity.

8. DEBUGGING AND TESTING

Since both designs multiplex the address and data words through the same 32 I/O pins, the actual processor does not use more than about 50 pins. The first prototype runs are being mounted in square 84-pin packages. Many extra pins have been devoted to debugging and testing. Important points of the chip are wired up in a scan-in/scan-out manner[Eich78] to permit loading or examining almost the complete machine state at will. To shorten the scan length through all these flip-flops, five separate loops were used in the RISC I chip, with access to them from the chip periphery.

While waiting for properly fabricated wafers to be returned to us, we built up a debugging station for the chips. Most of the students working on the RISC I project came from a computer science background and were thus more at home with sophisticated software than with oscilloscope and logic-state analyzer. This was taken into account when constructing our test set-up. The hardware consists of a microprocessor box with an attached terminal and a special socket for the chip to be tested. One RAM closely associated with the test socket forces patterns onto the input pins of the chip under the control of a hardware clock running at speeds up to 4 MHz. A second RAM receives the resulting bit patterns at the output pins. The driving patterns are prepared in the UNIX environment on the host computer (VAX 11/780) and are then down-loaded to the input RAM through the interface box containing a Z8000 microprocessor. The result patterns are uploaded from the output RAM through the microprocessor box to the host where they can be manipulated and analyzed in the software environment familiar to most of the users of this special test facility.

The diagnostic programs previously used for the simulation of the RISC design were used again to test the actual chips. This time SLANG was running in parallel with the tester, comparing the results accumulated in the result RAM with the patterns obtained from simulation.

The chip was equipped with scan-in/scan-out (SISO) hardware to allow separate testing of each major block. There are 5 SISO loops in RISC I: one each for the shifter, ALU input, ALU output, the program counters, and control. The first set of chips had faulty processing, and no electrical measurements could be made. The second set had a severe yield problem due to too narrow gaps between the poly-silicon features, but a few working SISO loops were found. However, the 80-bit long SISO loop through the control section never worked completely. Its operation is required for complete control of the other SISO loops. In hindsight it is clear that completely separate control pins for all five SISO loops would have been a much more prudent approach.

In the third batch of wafers no chip had all SISO loops operational, but two chips showed "signs of live" when tried with small test programs. Further testing showed that in spite of yield flaws, these chips performed most low-level functions as intended. A few bits of the data were stuck to 0 or 1, yet these chips could execute many of the instructions. Once it was known that the design was largely correct, the SISO loops were ignored and diagnostic programs were run on the chip. Even when entire wafers needed to be screened to determine which chips to mount (with the tester interfaced to the wafer prober), Foderaro and VanDyke relied on diagnostics programs rather than on tests involving the SISO loops.[Fode82]

Later Van Dyke designed and built a demonstration board for RISC I. It included the necessary systems elements around the CPU to make a complete computer, e.g. I/O, memory and memory management. A chip with some of the upper bits in the datapath stuck high successfully ran the first RISC I program on June 11, 1982, reading characters from a terminal, changing them according to a simple key, and writing them back out.

At this point we created new and more sophisticated diagnostic programs, – and we uncovered our first design error, associated with the optional setting of condition codes on the load and shift instructions. Fortunately, by making a suitable change in the RISC I assembler, we could "eliminate" this error. From the fourth batch of wafers we have tested about 40 chips and found four chips without functional flaws. This is quite a satisfactory yield for a 10.2 by 7.75 mm (406 × 305 mil) chip.

The fastest of these chips runs all diagnostics at a 1.5 MHz clock rate at room temperature, corresponding to 2 μsec per RISC I instruction. This is rather disappointing since we had expected this latter figure to be near 400 ns. Obvious critical paths (bus discharge, ALU carry chain, etc) were analyzed with SPICE and our circuits refined to meet that goal. However, there was no guarantee that we had looked at *all* possible critical paths. Indeed, some of our diagnostics can be run at a faster clock, indicating that many RISC I instructions are faster than 2 μsec, and that the problem may be very localized. Debugging of these performance limitations is currently under way.

It becomes clear from this experience that a most urgently needed tool for our design environment is a timing verifier. There are higher-level timing simulators running at Berkeley, but they were not closely enough integrated into the design environment readily familiar to the designers of RISC I. We need a program that can take a whole chip description in the same format as used for the functional or switch-level simulation and point out for all clock changes the speed-limiting path.

Another insight gained was that SISO loops are only useful in debugging the chip if the proper tests are prepared at design time. Had we written diagnostic patterns involving the SISO loops, we would also have noticed how cumbersome their usage is without separate control for each loop and might have changed the design. The scan loops are still expected to be worth-while in production testing to reduce the number of necessary tests.

9. EVALUATION OF THE RISC I ARCHITECTURE

To evaluate the RISC concept and the specific architecture chosen, we compared it to other computers, using a dozen *C* programs. The *C* compilers used were very similar; the VAX and RISC compilers are both based on the UNIX Portable *C* Compiler[John78] and the one for the PDP-11 is based on the Ritchie *C* compiler.[Ritc75]

Our most detailed studies were done on *puzzle* and *qsort*. *Puzzle*, developed by Forest Baskett, is a recursive bin-packing program that solves a three-dimensional puzzle. It displays many features of typical programs, except that it has relatively few procedure calls. Nevertheless, the nesting depth of the procedures reaches 20. There is a relatively large number of loops. *Qsort* is a recursive quicksort program. In our test runs, this program sorts 2600 fixed length character strings. The somewhat unusual feature of this program is that it has a relatively high incidence of memory references. The execution of this program results in 1713 multiplies and 1712 divides which are subroutines in RISC I.

In a comparison of the static number of instructions and static size of programs we found that on the average RISC uses only two thirds more instructions than the VAX and about two fifths more than the PDP-11, in spite of the fact that RISC I has only very 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 code density optimization was virtually ignored.

Our main goal for RISC I was to obtain good performance for high-level programs. The main performance comparison was based on the execution times of these and other programs on various computers. While we could directly measure execution times on other machines, for RISC I we originally had to rely on simulation. For these simulations the instruction set of Table 2 was used and a machine cycle of 400 nanoseconds was assumed. This estimate is based on extensive circuit simulation of the critical sections in our data path, which yield about 100 nsec to read one of 138 registers, 200 nsec to perform a 32-bit addition, and 100 nsec to store the result back into the register file. Even though the first layout of RISC I did not achieve this performance because of some rather long signal paths that were originally overlooked, the above numbers for the cycle time must be considered quite conservative and can easily be met in a carefully debugged design.

Table 4 shows the code size and execution time of six *C* programs on three different computers. The VAX 11/780 is a 32-bit Schottky-TTL minicomputer with a 200 ns microcycle time; and the Z8002 is a 16-bit NMOS microprocessor with a microcycle time of 250 ns. In comparison with the Z8002, which is using only 16-bit addresses and data, RISC I programs are typically 10% larger but run about four times faster. The byte-variable length of the VAX instructions reduces program size by about a third; but, much to our surprise, for every *C* program that we have run, the RISC I simulation has outperformed the VAX 11/780.

We can identify several contributing factors to this surprisingly high performance of RISC I. A strong contribution is due to the overlapped banks of

Table 4.
RISC I Program Sizes and Execution Times Relative to a VAX 11/780 and a Z8000

Name	Program Size (bytes)					Execution Time (secs)				
	RISC	VAX	RISC rel	Z8000	RISC rel	RISC	VAX	RISC rel	Z8000	RISC rel
acker	208	120	1.73	238	0.87	3.2	5.1	.63	8.8	.36
qsort	644	436	1.48	648	0.99	0.8	1.8	.44	4.7	.17
puzzle(sub)	2468	1668	1.48	1612	1.53	4.7	9.5	.49	19.2	.24
puzzle(ptr)	2480	1700	1.46	1656	1.50	3.2	4.0	.80	7.5	.43
sed	17368	14336	1.21	17500	0.99	5.1	5.7	.89	22.2	.23
towers	132	100	1.32	242	0.55	6.8	12.2	.56	28.7	.24
Average	3883	3060	1.5 ±.2	3649	1.1 ±.3	4.0	6.4	.6 ±.2	15.2	.3 ±.1

registers.[Patt81, Patt82c] As indicated by the two benchmark programs used in Table 5, the overlapped register banks have been effective in reducing the cost of using procedures. The puzzle and quicksort programs discussed above span quite a range in their percentage of procedure calls. While puzzle makes heavy use of FOR-loops, quicksort is a very recursive program. Table 5 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 I 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 the processor has the desired set of eight register windows and that half of the registers are saved on an overflow and half are restored on an underflow. We find that for RISC I on the average only 0.37 words are transferred to memory per procedure invocation in the puzzle program, while this number is 0.07 for quicksort. On the VAX these numbers are 20.7 words and 12.6 words, respectively. In other terms, note that half of the data memory references in *qsort* are the result of the call/return overhead of the VAX.

The multiple register banks in RISC I have allowed the allocation of local variables in registers. 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. In traditional machines, generally 30 to 50% of the instructions access data memory, and less than 20% of the instructions are of the register-to-register type.[Alex75] 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.[Patt82b] The reduced number of off-chip memory accesses, is also responsible for the improved performance of RISC.

Table 5.
Memory Traffic Due to Call/Return

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

Another reason why RISC is outperforming the VAX is that the existing *C* compilers for the VAX are not able to exploit the existing architecture effectively. It appears that the complexity of the hardware of most modern computers has outpaced the abilities of present-day compilers. The RISC I architecture, on the other hand, is so simple that even a straightforward compiler is effective. Comparisons of hand-optimized assembler-level programs with the corresponding HLL ones show indeed that on the VAX a lot can be gained by going to assembler code, whereas for RISC the corresponding gain is quite small.

RISC I does not support floating point instructions. Since not every application needs them, it appears an unnecessary burden for a single-chip general-purpose CPU to provide these instructions. The proper approach seems to be the one now followed by many manufacturers of microprocessors: to add co-processors for all such special functions.

10. RELATED MACHINES AND CONCEPTS

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. A popular architecture that is close to the RISC concept is represented by the Texas Instruments 990-9900 family. These machines allocate their general registers in memory, with a single register pointing to this work space. Adding the contents of one register to another results in three memory accesses. 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. [Orla81] The machine that came closest to having 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 same chip; the MAC-8 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.

Most modern machines support procedure calls by having instructions that manage a portion of main memory as a stack on which parameters are passed and locals are allocated. Thus, as an alternative we have considered a memory-to-memory architecture enhanced with a sizable cache. However, a cache is not only harder to implement, but also slower.

First, with a cache a full address calculation (stack pointer plus offset) is required for every register access. The virtual address translation and decoding would also be slower than a direct register access. Further, a cache is ineffective if it is too small. An effective data cache would require a much larger area than the RISC I register file, especially if it must provide multiple ports to each location. A bigger memory structure will also have longer and slower address lines and data busses. Finally, the more complicated cache control would have extended the design phase of RISC I.

11. CONCLUSIONS

From our experience with two designs and the simulation of several small programs, we are convinced that Reduced Instruction Set Computers show a promising way of processor design in general and a very good match to VLSI implementation. We have taken out most of the complexity of modern computers with only a moderate loss in code density and even a gain in performance. These simplifications have not reduced the functionality of RISC I & II; the chosen subset of instructions is sufficient to compile high-level language programs into code that will execute them correctly and efficiently.

While RISC I has substantially reduced the number of *data* accesses in all programs, the number of *instruction* accesses has increased. This is due in part to the number of NOP's introduced, and in part due to the inefficient, fixed-size encoding of the instructions. It is clear that successors to RISC I will have to address the issue of code density.

The overlapped register banks make a significant contribution towards the performance of RISC I & II. They effectively provide the function of a cache to alleviate the impact of the speed mismatch of the internal on-chip circuitry in the processor and the cycle times of external memory parts. For the same reason, the RISC I architecture also requires an instruction cache to work with full efficiency.

The dramatically reduced size of the control section of RISC I increases overall regularity of the chip and thus reduces the amount of time spent on layout. RISC's will thus have a significantly shorter development cycle and are more likely to emerge without design flaws.

12. ACKNOWLEDGEMENTS

The RISC investigation started with a four quarter sequence of graduate courses at Berkeley. Many students have participated and several of them have made significant contributions. Manolis Katevenis designed the initial block structure, the initial timing description, and provided many important ideas for the implementation and the architecture. Ralph Campbell wrote the initial C compiler, the optimizer, the assembler, and the linker. Yuval Tamir wrote a simulator, ran many benchmarks,[Tami81] and participated in the initial design of

RISC I. Gary Corcoran wrote the first ISPS description of RISC I. Jim Peek, Korbin Van Dyke, John Foderaro, Dan Fitzpatrick, and Zvi Peshkess were the principal VLSI designers of the RISC I chip. Manolis Katevenis and Bob Sherburne have designed the more sophisticated RISC II chip. Michael Arnold, Dan Fitzpatrick, John Foderaro, and Howard Landman all wrote CAD tools that were crucial to the implementation of the RISC chips. Peter Kessler contributed to the development of the overlapped register window scheme and to our CAD software. Jim Beck and Bob Cmelik created the VLSI testing hardware and software. Earl Cohen and Neil Soiffer collected statistics on C programs and Shafi Goldwasser collected similar statistics for Pascal.

The authors would like to thank Manolis Katevenis and Robert Sherburne for their contributions to this paper.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251.

References

Alex75. Alexander, W.C. and Wortman, D.B., "Static and Dynamic Characteristics of XPL Programs," *Computer* 8(11) pp. 41-46 (Nov. 1975).

Allm81. Allman, E., "An Introduction to the Source Code Control System," Technical Memo, U.C. Berkeley (Jan. 1981).

Arno82. Arnold, M.H. and Ousterhout, J.K., "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 19th Des. Autom. Conf.*, Las Vegas, pp. 530-536 (June 1982).

Bake80a. Baker, C.M. and Terman, C., "Tools for Verifying Integrated Circuit Designs," *Lambda* 1(3) pp. 22-30 (4th Q. 1980).

Bake80b. Baker, C.M., "Artwork Analysis Tools for VLSI Circuits," TR-239, Lab. for Comp. Science, M.I.T. (May 1980).

Bask78. Baskett, F., *A VLSI Pascal Machine*, Puolic Lecture, University of California, Berkeley, Fall 1978.

Bell70. Bell, G. and Newell, A., "The PMS and ISP Descriptive System for Computer Systems," *Proc. AFIPS SJCC*, (1970).

Camp81. Campbell, R., "A Peephole Optimizer for RISC," Internal Memo, U.C. Berkeley (1981).

Cohe82. Cohen, D. and Tyree, V., "Quality Control from the Silicon Brokers Perspective," *VLSI Design* III(4) pp. 24-30 (July 1982).

Eich78. Eichelberger, E.B. and Williams, T.W., "A Logic Design Structure for LSI Testability," *Jour. Design Automation and Fault-Tolerant Computing* **2** pp. 165-178 (May 1978).

Fitz81. 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 VanDyke, K.S., "VLSI Implementation of a Reduced Instruction Set Computer," *Proc. CMU Conf. on VLSI Systems and Computations*, Pittsburgh PA, pp. 327-336 (Oct. 1981).

Fitz82. Fitzpatrick, D.T., "MEXTRA: A Manhattan Circuit Extractor," ERL Memo M82/42, U.C. Berkeley (May 1982).

Fode81. Foderaro, J.K. and VanDyke, K.S., "SLANG Slinger's Cyclopedia," Internal Report, U.C. Berkeley (Dec. 1981).

Fode82. Foderaro, J.K., VanDyke, K.S., and Patterson, D.A., "Running RISC's," *VLSI Design* III(5) pp. 27-32 (Sept. 1982).

Henn81. Hennessy, J., Jouppi, N., Baskett, F., and Gill, J., "MIPS: A VLSI Processor Architecture," *Proc. CMU Conf. on VLSI Systems and Computations*, Pittsburgh PA , pp. 337-346 (Oct. 1981).

Henn82. Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J., "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symp. on Architectural Support for Prog. Lang. and Oper. Systems*, Palo Alto, CA, pp. 2-11 (March 1982).

John78. Johnson, S.C., "A Portable Compiler: Theory and Practice," *Proc. Fifth Annual ACM Symposium of Programming Languages*, Tuscon, Arizona, pp. 97-104 (Jan. 1978).

Latt81. Lattin, W.W., Bayliss, J.A., Budde, D.L., Rattner, J.R., and Richardson, W.S., "A Methodology for VLSI Chip Design," *Lambda* 2(2) pp. 34-44 (2nd Q. 1981).

Lyon81. Lyon, R.F., "Simplified Design Rules for VLSI Layouts," *Lambda* 2(1) pp. 54-59 (1st Q. 1981).

Morr79. Morris, D. and Ibbett, R. N., *The MU-5 Computer System*, Springer Verlag (1979).

Orla81. Orlando, R.V. and Anderson, T.L., "An Overview of the 9900 Microprocessor Family," *IEEE MICRO* 1(3) pp. 38-42 (Aug. 1981).

Oust81. Ousterhout, J., "Caesar: An Interactive Editor for VLSI Circuits," Internal Memo, U.C. Berkeley (July 1981).

Patt80a. Patterson, D.A. and Ditzel, D.R., "The Case for the Reduced Instruction Set Computer," *Computer Architecture News* 8(6) pp. 25-33 (Oct. 1980).

Patt80b. Patterson, D.A. and Séquin, C.H., "Design Considerations for Single-Chip Computers of the Future," *IEEE Trans. on Computers* C-29(2) pp. 108-116 (Feb. 1980). Joint Spec. Issue w. IEEE Jour. Solid-State Circuits.

Patt81. Patterson, D.A. and Séquin, C.H., "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. 8th Intnl. Symp. on Computer Arch.*, Minneapolis, MINN, pp. 443-457 (May 1981).

Patt82a. Patterson, D.A., Garrison, P., Hill, M., Lioupis, D., Nyberg, C., Sippel, T., and VanDyke, K., "Architecture of a VLSI Instruction Cache," Internal Report, U.C.Berkeley (1982). Submitted to 10th Intnl. Symp. on Computer Architecture.

Patt82b. Patterson, D.A. and Piepho, R.S., "RISC Assessment: A High-Level Language Experiment," *Proc. 9th Intnl. Symp. on Computer Architecture*, Austin, Texas, pp. 3-8 (April 1982).

Patt82c. Patterson, D.A. and Séquin, C.H., "A VLSI RISC," *Computer* 15(9) pp. 8-21 (Sept. 1982).

Radi82. Radin, G., "The 801 Minicomputer," *Proc. Symp. on Architectural Support for Prog. Lang. and Oper. Systems*, Palo Alto, CA, pp. 39-47 (March 1982).

Ritc75. Ritchie, D.M., *A Tour through the UNIX C Compiler*, (unpublished) 1975.

Sher82. Sherburne, R.W., Katevenis, M.G.H., Patterson, D.A., and Séquin, C.H., "Datapath Design for RISC," *Proc. Conf. on Adv. Research in VLSI*, M.I.T., Cambridge, MA, pp. 53-62 (Jan. 1982).

Site79. Sites, R.L., "How to Use 1000 Registers," *Proc. Caltech Conf. on VLSI*, Pasadena, CA, pp. 527-532 (Jan. 1979).

Spro80. Sproull, R.F. and Lyon, R.F., "The Caltech Intermediate Form for LSI Layout Description," pp. 115-127 in *Introduction to VLSI Systems*, ed. C.A. Mead and L.A. Conway, Addison-Wesley, Reading, Mass. (1980).

Tami81. Tamir, Y., "Simulation and Performance Evaluation of the RISC Architecture," ERL Memo M81/17, U.C. Berkeley (March 1981).

Tami82. Tamir, Y. and Séquin, C.H., "Strategies for Managing the Register File in RISC," *submitted to IEEE Trans. on Computers*, (July 1982).

Term82. Terman, C., "Simulation Tools for VLSI Design," Thesis, Lab. for Comp. Science, M.I.T. (1982).

VanD82. VanDyke, K.S., "SLANG: A Logic Simulation Language," Master's Report, U.C. Berkeley (June 1982).