REDUCED INSTRUCTION SET COMPUTERS

Reduced instruction set computers aim for both simplicity in hardware and synergy between architectures and compilers. Optimizing compilers are used to compile programming languages down to instructions that are as unencumbered as microinstructions in a large virtual address space, and to make the instruction cycle time as fast as possible.

DAVID A. PATTERSON

As circuit technologies reduce the relative cost of processing and memory, instruction sets that are too complex become a distinct liability to performance. The designers of reduced instruction set computers (RISCs) strive for both simplicity in hardware and synergy between architecture and compilers, in order to streamline processing as much as possible. Early experience indicates that RISCs can in fact run much faster than more conventionally designed machines.

BACKGROUND

The IBM System/360, first introduced in 1964, was the real beginning of modern computer architecture. Although computers in the System/360 "family" provided a different level of performance for a different price, all ran identical software. The System/360 originated the distinction between *computer architecture*—the abstract structure of a computer that a machine-language programmer needs to know to write programs—and the *hardware implementation* of that structure. Before the System/360, architectural trade-offs were determined by the effect on price and performance of a single implementation; henceforth, architectural trade-offs became more esoteric. The consequences of single implementations would no longer be sufficient to settle an argument about instruction set design.

Microprogramming was the primary technological innovation behind this marketing concept. Microprogramming relied on a small control memory and was an elegant way of building the processor control unit for a large instruction set. Each word of control memory is called a *microinstruction*, and the contents are essentially an interpreter, programmed in microinstructions. The main memories of these computers were magnetic core memories, the small control memories of which were usually 10 times faster than core.

Minicomputer manufacturers tend to follow the lead of mainframe manufacturers, especially when the mainframe manufacturer is IBM, and so microprogramming caught on quickly. The rapid growth of semiconductor memories also speeded this trend. In the early 1970s, for example, 8192 bits of read-only memory (ROM) took up no more space than 8 bits of register. Eventually, minicomputers using core main memory and semiconductor control memory became standard in the minicomputer industry.

With the continuing growth of semiconductor memory, a much richer and more complicated instruction set could be implemented. The architecture research community argued for richer instruction sets. Let us review some of the arguments they advanced at that time:

1. *Richer instruction sets would simplify compilers.* As the story was told, compilers were very hard to build, and compilers for machines with registers were the hardest of all. Compilers for architectures with execution models based either on stacks or memory-to-memory operations were much simpler and more reliable.

2. *Richer instruction sets would alleviate the software crisis.* At a time when software costs were rising as fast as hardware costs were dropping, it seemed

appropriate to move as much function to the hardware as possible. The idea was to create machine instructions that resembled programming language statements, so as to close the "semantic gap" between programming languages and machine languages.

3. *Richer instruction sets would improve architecture quality.* After IBM differentiated architecture from implementation, the research community looked for ways to measure the quality of an architecture, as opposed to the speed at which implementations could run programs. The only architectural metrics then widely recognized were program size, the number of bits of instructions, and bits of data fetched from memory during program execution (see Figure 1).

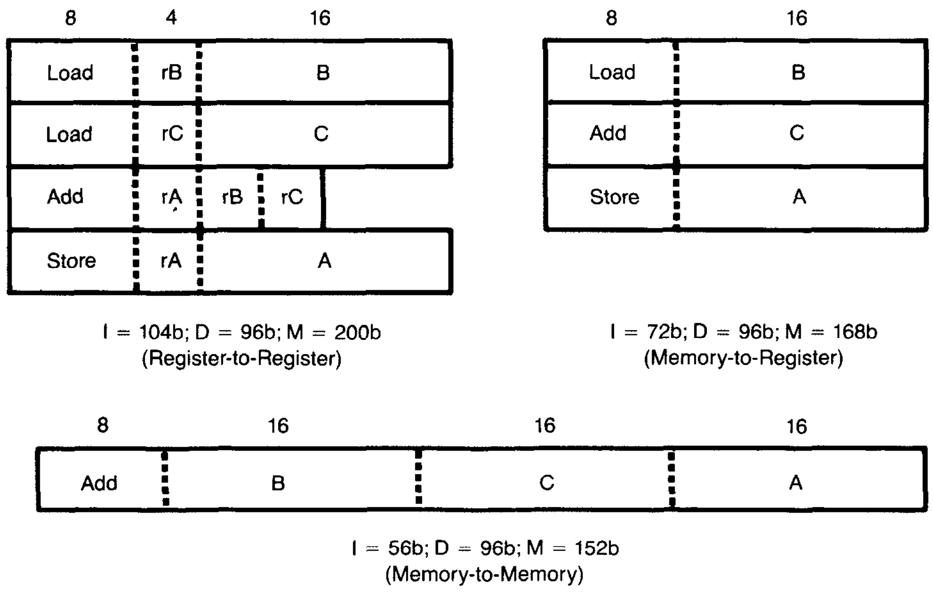
Memory efficiency was such a dominating concern in these metrics because main memory—magnetic core memory—was so slow and expensive. These metrics are partially responsible for the prevailing belief in the 1970s that execution speed was proportional to program size. It even became fashionable to examine long lists of instruction execution to see if a pair or triple of old instructions could be replaced by a single, more powerful instruction. The belief that larger programs were invariably slower programs inspired the invention of many exotic instruction formats that reduced program size.

The rapid rise of the integrated circuit, along with arguments from the architecture research community in the 1970s led to certain design principles that guided computer architecture:

1. *The memory technology used for microprograms was growing rapidly, so large microprograms would add little or nothing to the cost of the machine.*
2. *Since microinstructions were much faster than normal machine instructions, moving software functions to microcode made for faster computers and more reliable functions.*
3. *Since execution speed was proportional to program size, architectural techniques that led to smaller programs also led to faster computers.*
4. *Registers were old fashioned and made it hard to build compilers; stacks or memory-to-memory architectures were superior execution models. As one architecture researcher put it in 1978, "One's eyebrows should rise whenever a future architecture is developed with a register-oriented instruction set."*[1]

Computers that exemplify these design principles are the IBM 370/168, the DEC VAX-11/780, the Xerox

[1] Myers, G.J. The case against stack-oriented instruction sets. *Comput. Archit. News* 6, 3 (Aug. 1977), 7–10.



In this example, the three data words are 32 bits each and the address field is 16 bits. Metrics were selected by research architects for deciding which architecture is best; they selected the total size of executed instructions (I), the total size of executed data (D), and the total memory traffic—that is, the sum of I and D, which is (M).

These metrics suggest that a memory-to-memory architecture is the "best" architecture, and a register-to-register architecture the "worst." This study led one research architect in 1978 to suggest that future architectures should not include registers.

FIGURE 1. The Statement A ⟵ B + C Translated into Assembly Language for Three Execution Models: Register-to-Register, Memory-to-Register, and Memory-to-Memory

TABLE I. Four Implementations of Modern Architectures

	IBM 370/168	VAX-11/780	Dorado	iAPX-432
Year	1973	1978	1978	1982
Number of instructions	208	303	270	222
Control memory size	420 Kb	480 Kb	136 Kb	64 Kb
Instruction sizes (bits)	16–48	16–456	8–24	6–321
Technology	ECL MSI	TTL MSI	ECL MSI	NMOS VLSI
Execution model	reg-mem	reg-mem	stack	stack
	mem-mem	mem-mem		mem-mem
	reg-reg	reg-reg		
Cache size	64 Kb	64 Kb	64 Kb	0

These four implementations, designed in the 1970s, all used microprogramming. The emphasis on memory efficiency at that time led to the varying-sized instruction formats of the VAX and the 432. Note how much larger the control memories were than the cache memories.

Dorado, and the Intel iAPX-432. Table I shows some of the characteristics of these machines.

Although computer architects were reaching a consensus on design principles, the implementation world was changing around them:

- Semiconductor memory was replacing core, which meant that main memories would no longer be 10 times slower than control memories.
- Since it was virtually impossible to remove all mistakes for 400,000 bits of microcode, control store ROMs were becoming control store RAMs.
- Caches had been invented—studies showed that the locality of programs meant that small, fast buffers could make substantial improvement in the implementation speed of an architecture. As Table I shows, caches were included in nearly every machine, though control memories were much larger than cache memories.
- Compilers were subsetting architectures—simple compilers found it difficult to generate the complex new functions that were included to help close the "semantic gap." Optimizing compilers subsetted architectures because they removed so many of the unknowns at compiler time that they rarely needed the powerful instructions at run time.

WRITABLE CONTROL STORE

One symptom of the general dissatisfaction with architectural design principles at this time was the flurry of work in writable control memory, or writable control store (WCS). Researchers observed that microcoded machines could not run faster than 1 microcycle per instruction, typically averaging 3 or 4 microcycles per instruction; yet the simple operations in many programs could be found directly in microinstructions. As long as machines were too complicated to be implemented by ROMs, why not take advantage of RAMs by loading different microprograms for different applications?

One of the first problems was to provide a programming environment that could simplify the task of writing microprograms, since microprogramming was the most tedious form of machine-language programming. Many researchers, including myself, built compilers and debuggers for microprogramming. This was a formidable assignment, for virtually no inefficiencies could be tolerated in microcode. These demands led to the invention of new programming languages for microprogramming and new compiler techniques.

Unfortunately for me and several other researchers, there were three more impediments that kept WCSs from being very popular. (Although a few machines offer WCS as an option today, it is unlikely that more than one in a thousand programmers take this option.) These impediments were

1. *Virtual memory complications.* Once computers made the transition from physical memory to virtual memory, microprogrammers incurred the added difficulty of making sure that any routine could start over if any memory operand caused a virtual memory fault.
2. *Limited address space.* The most difficult programming situation occurs when a program must be forced to fit in too small a memory. With control memories of 4096 words or less, some unfortunate WCS developers spent more time squeezing space from the old microcode than they did writing the new microcode.
3. *Swapping in a multiprocess environment.* When each program has its own microcode, a multiprocess operating system has to reload the WCS on each process switch. Reloading time can range from 1,000 to 25,000 memory accesses, depending on the machine. This added overhead decreased the performance benefits gained by going to a WCS in the first place.

These last two difficulties led some researchers to conclude that future computers would have to have virtual control memory, which meant that page faults could occur during microcode execution. The distinction between programming and microprogramming was becoming less and less clear.

THE ORIGINS OF RISCS

About this point, several people, including those who had been working on microprogramming tools, began to rethink the architectural design principles of the 1970s. In trying to close the "semantic gap," these principles had actually introduced a "performance gap." The attempt to bridge this gap with WCSs was unsuccessful, although the motivation for WCS—that instructions should be no faster than microinstructions and that programmers should write simple operations that map directly onto microinstructions—was still valid. Furthermore, since caches had allowed "main" memory accesses at the same speed as control memory accesses, microprogramming no longer enjoyed a ten-to-one speed advantage.

A new computer design philosophy evolved: Optimizing compilers could be used to compile "normal" programming languages down to instructions that were as unencumbered as microinstructions in a large virtual address space, and to make the instruction cycle time as fast as the technology would allow. These machines would have fewer instructions—a *reduced set*—and the remaining instructions would be simple and would generally execute in one cycle—*reduced instructions*—hence the name *reduced instruction set computers* (RISCs). RISCs inaugurated a new set of architectural design principles:

1. *Functions should be kept simple unless there is a very good reason to do otherwise.* A new operation that increases cycle time by 10 percent must reduce the number of cycles by at least 10 percent to be worth considering. An even greater reduction might be necessary, in fact, if the extra development effort and hardware resources of the new function, as they impact the rest of the design, are taken into account.
2. *Microinstructions should not be faster than simple instructions.* Since cache is built from the same technology as writable control store, a simple instruction should be executed at the same speed as a microinstruction.
3. *Microcode is not magic.* Moving software into microcode does not make it better, it just makes it harder to change. To paraphrase the Turing Machine argument, *anything that can be done in a microcoded machine can be done in assembly language in a simple machine.* The same hardware primitives assumed by the microinstructions must be available in assembly language. The run-time library of a RISC has all the characteristics of a function in microcode, except that it is easier to change.
4. *Simple decoding and pipelined execution are more important than program size.* Imagine a model in which the total work per instruction is broken into pieces, and different pieces for each instruction execute in parallel. At the peak rate a new instruction is started every cycle (Figure 2). This assembly-line approach performs at the rate determined by the length of individual pieces rather than by the total length of all pieces. This kind of model gave rise to instruction formats that are simple to decode and to pipeline.
5. *Compiler technology should be used to simplify instructions rather than to generate complex instructions.* RISC compilers try to remove as much work as possible at compile time so that simple instructions can be used. For example, RISC compilers try to keep operands in registers so that simple register-to-register instructions can be used. Traditional compilers, on the other hand, try to discover the ideal addressing mode and the shortest instruction format to add the operands in memory. In general, the designers of RISC compilers prefer a register-to-register model of execution so that compilers can keep operands that will be reused in registers, rather than repeating a memory access or a calculation. They therefore use LOADs and STOREs to access memory so that operands are not implicitly discarded after being fetched, as in the memory-to-memory architecture (see Figure 3).

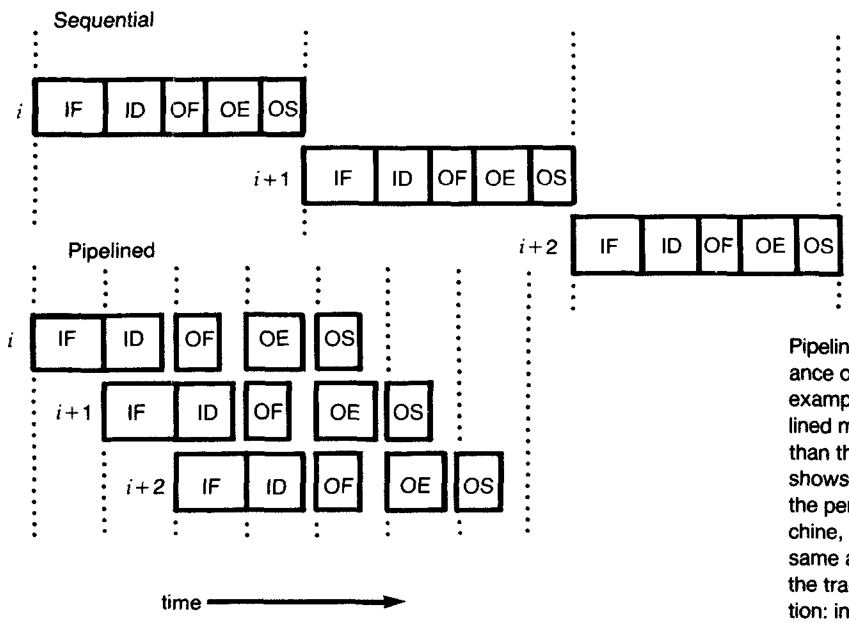
COMMON RISC TRAITS

We can see these principles in action when we look at some actual RISC machines: the 801 from IBM Research, the RISC I and the RISC II from the University of California at Berkeley, and the MIPS from Stanford University.

All three RISC machines have actually been built and are working (see Figure 4). The original IBM 801 was built using off-the-shelf MSI ECL, whereas the Berkeley RISCs and the Stanford MIPS were built with custom NMOS VLSI. Table II shows the primary characteristics of these three machines.

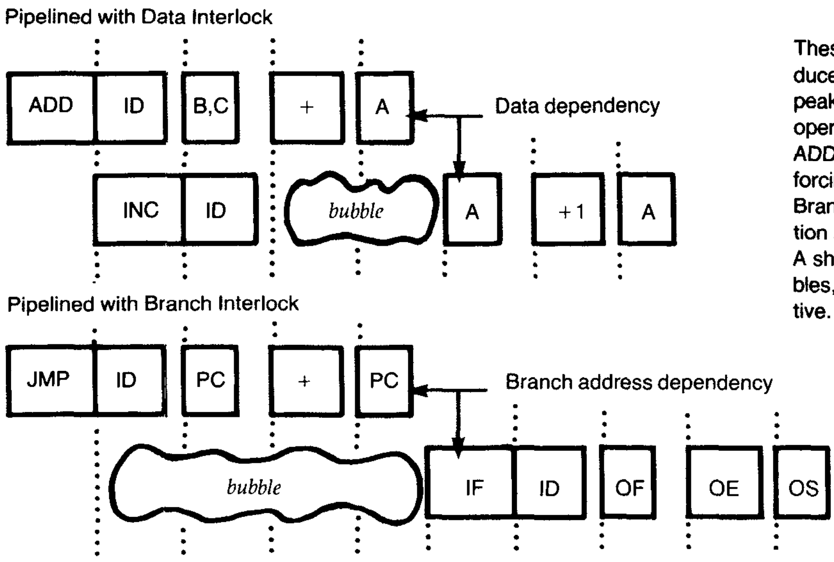
Although each project had different constraints and goals, the machines they eventually created have a great deal in common:

1. *Operations are register-to-register, with only LOAD and STORE accessing memory.* Allowing compilers to reuse operands requires registers. When only LOAD and STORE instructions access memory, the instruction set, the processor, and the handling of page faults in a virtual memory environment are greatly simplified. Cycle time is shortened as well.
2. *The operations and addressing modes are reduced.* Operations between registers complete in one cycle, permitting a simpler, hardwired control for each RISC, instead of microcode. Multiple-cycle instructions such as floating-point arithmetic are either executed in software or in a special-purpose coprocessor. (Without a coprocessor, RISCs have mediocre floating-point performance.) Only two simple addressing modes, indexed and PC-relative, are provided. More complicated addressing modes can be synthesized from the simple ones.
3. *Instruction formats are simple and do not cross word boundaries.* This restriction allows RISCs to re-



Pipelined execution gives a peak performance of one instruction every step, so in this example the peak performance of the pipelined machine is about four times faster than the sequential version. This figure shows that the longest piece determines the performance rate of the pipelined machine, so ideally each piece should take the same amount of time. The five pieces are the traditional steps of instruction execution: instruction fetch (IF), instruction decode (ID), operand fetch (OF), operand execution (OE), and operand store (OS).

FIGURE 2a. Execution of Three Instructions for Sequential and Pipelined Execution

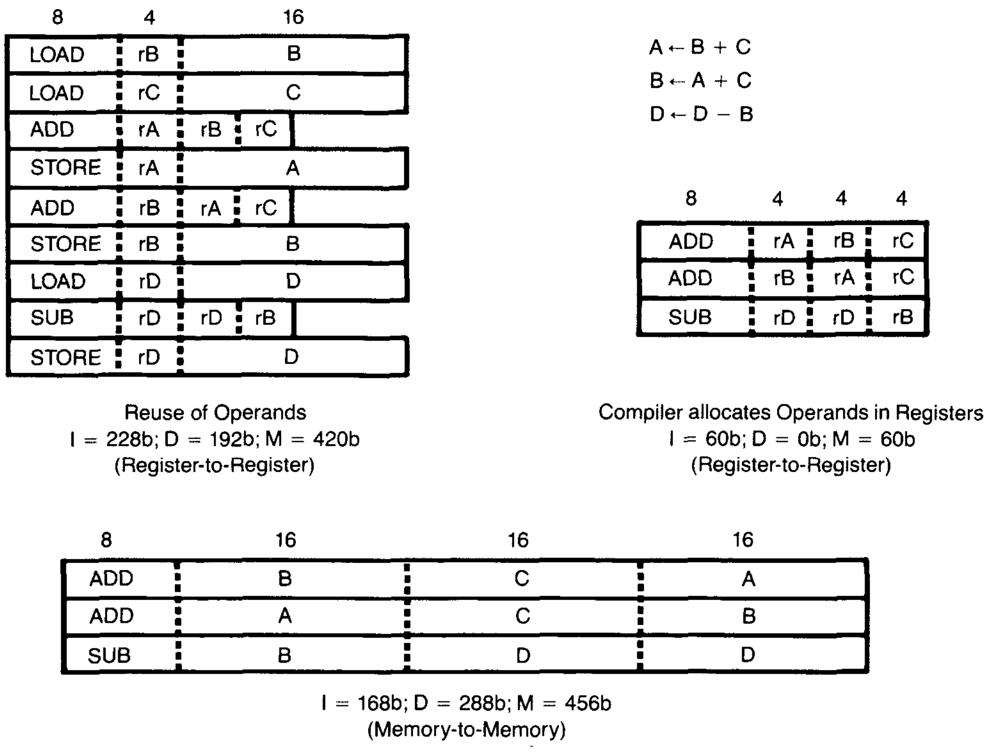

Pipelined with Data Interlock
ADD
ID
B,C
+
A
Data dependency
INC
ID
bubble
A
+1
A
Pipelined with Branch Interlock
JMP
ID
PC
+
PC
Branch address dependency
bubble
IF
ID
OF
OE
OS

These branches and data dependencies reduce the average instruction rate to below peak. The instruction INC A cannot fetch operand A until the previous instruction ADD B, C, A finishes storing B + C into A, forcing a two-stage bubble in the pipeline. Branches delay the pipeline until the instruction at the branch address can be fetched. A shorter pipeline would also shorten bubbles, thereby making pipelining more effective.

FIGURE 2b. Branches and Data Dependencies between Instructions Force Delays, or "Bubbles," into Pipelines

move instruction decoding time from the critical execution path. As Figure 5 shows, the Berkeley RISC register operands are always in the same place in the 32-bit word, so register access can take place simultaneously with opcode decoding. This removes the instruction decoding stage from the pipelined execution, making it more effective by shortening the pipeline (Figure 2b). Single-sized instructions also simplify virtual memory, since they cannot be broken into parts that might wind up on different pages.

4. *RISC branches avoid pipeline penalties.* A branch instruction in a pipelined computer will normally delay the pipeline until the instruction at the branch address is fetched (Figure 2b). Several pipelined machines have elaborate techniques for fetching



This new version assumes optimizing compiler technology for the sequence A ← B + C; B ← A + C; D ← D − B. Note that the memory-to-memory architecture has no temporary storage, which means that it must reload operands. The register-to-register architecture now appears to be the "best."

FIGURE 3. A New Version of the Architectural Metrics Presented in Figure 1

the appropriate instruction after the branch, but these techniques are too complicated for RISCs. The generic RISC solution, commonly used in microinstruction sets, is to redefine jumps so that they do not take effect until after the *following* instruction; this is called the *delayed branch.* The delayed branch allows RISCs to always fetch 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 RISCs are designed to be programmed in high-level languages, the programmer is not required to consider this issue; the "burden" is carried by the programmers of the compiler, the optimizer, and the debugger. The delayed branch also removes the branch bubbles normally associated with pipelined execution (see Figure 2b). Table III illustrates the delayed branch.

RISC optimizing compilers are able to successfully rearrange instructions to use the cycle after the delayed branch more than 90 percent of the time. Hennessy has found that more than 20 percent of all instructions are executed in the delay after the branch.

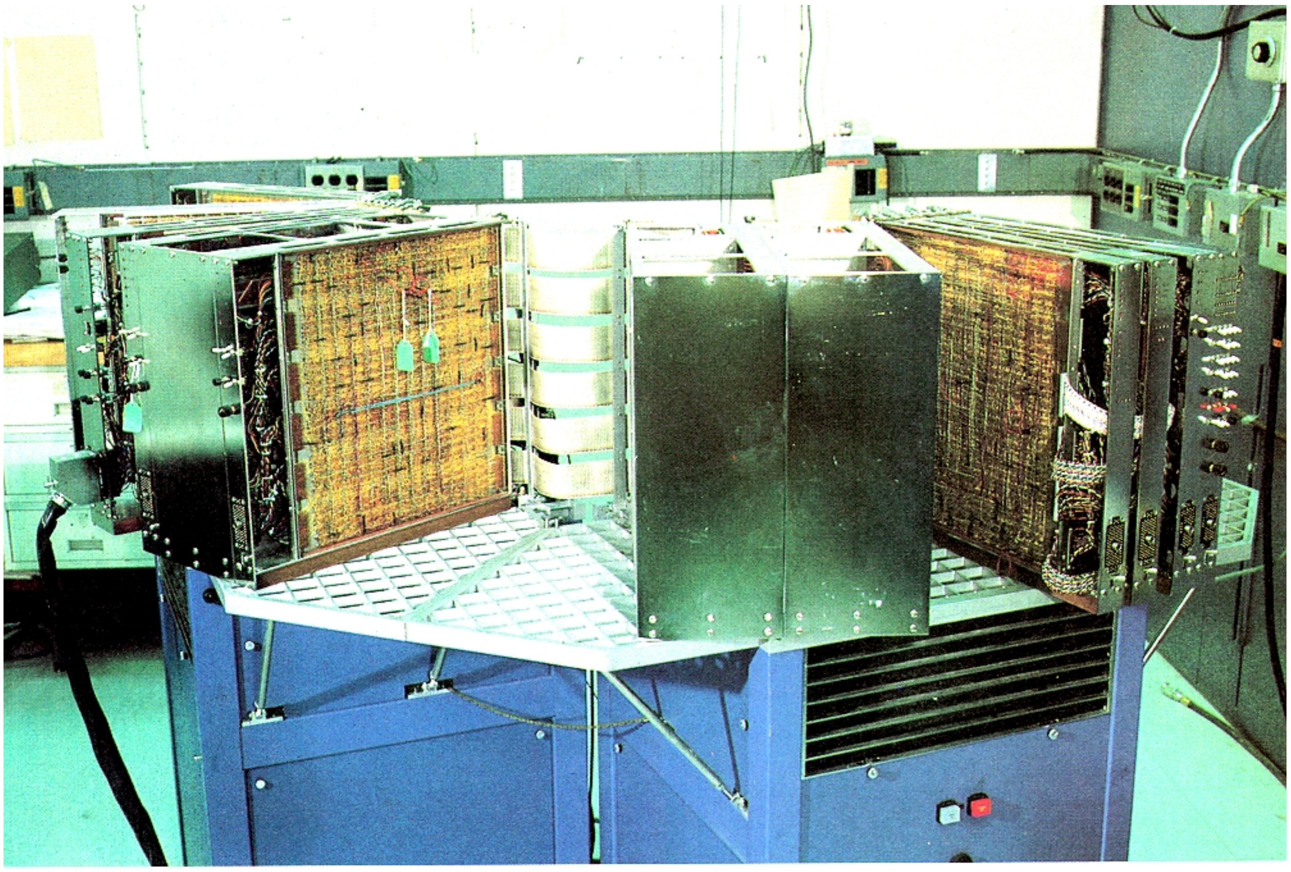
Hennessy also pointed out how the delayed branch illustrates once again the folly of architectural metrics. Virtually all machines with variable-length instructions use a buffer to supply instructions to the CPU. These units blindly fill the prefetch buffer no matter what instruction is fetched and thus load the buffer with instructions after a branch, despite the fact that these instructions will eventually be discarded. Since studies show that one in four VAX instructions changes the program counter, such variable-length instruction machines really fetch about 20 percent more instruction words from memory than the architecture metrics would suggest. RISCs, on the other hand, nearly always execute something useful because the instruction is fetched after the branch.

RISC VARIATIONS

Each RISC machine provides its own particular variations on the common theme. This makes for some interesting differences.

Compiler Technology versus Register Windows

Both IBM and Stanford pushed the state of the art in compiler technology to maximize the use of registers. Figure 6 shows the graph-coloring algorithm that is the cornerstone of this technology.



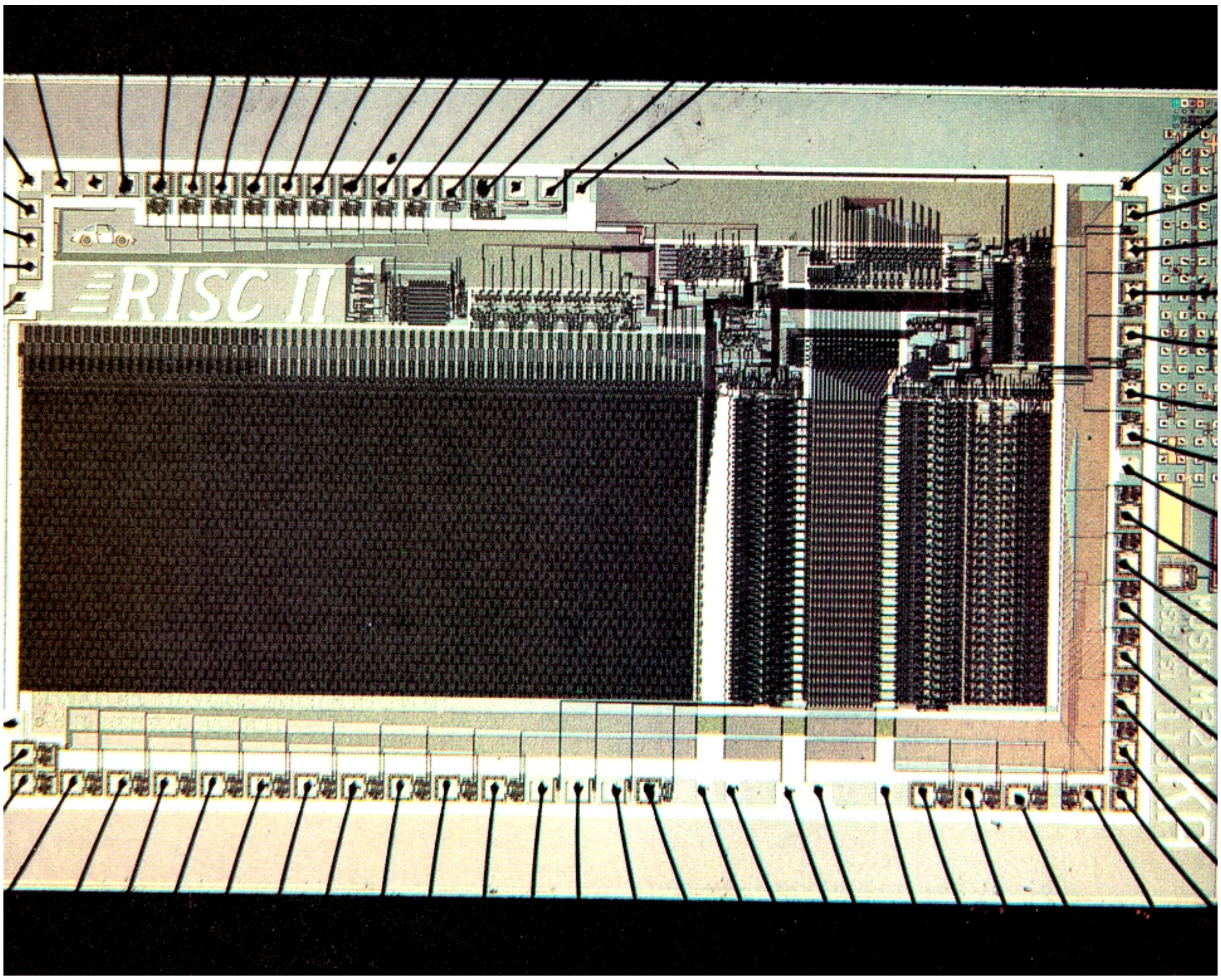
The IBM 801, above, was completed in 1979 and had a cycle time of 66 ns. It was built from off-the-shelf ECL. The RISC II, on the right, designed by Manolis Katevenis and Robert Sherburne, was designed in a four-micron single-level metal custom NMOS VLSI. This 41,000-transistor chip worked the first time at 500 ns per 32-bit instruction. The small control area accounts for only 10 percent of the chip and is found in the upper right-hand corner. The RISC II was rescaled and fabricated in three-micron NMOS. The resulting chip is 25 percent smaller than a 68000 and ran on first silicon at 330 ns per instruction. The Stanford MIPS, shown on page 16, chip was fabricated using the same conservative NMOS architecture and runs at 500 ns per instruction. This 25,000-transistor chip has about the same chip area as the RISC II. On-chip memory management support plus the two-instruction-per-word format increase the control area of the MIPS.

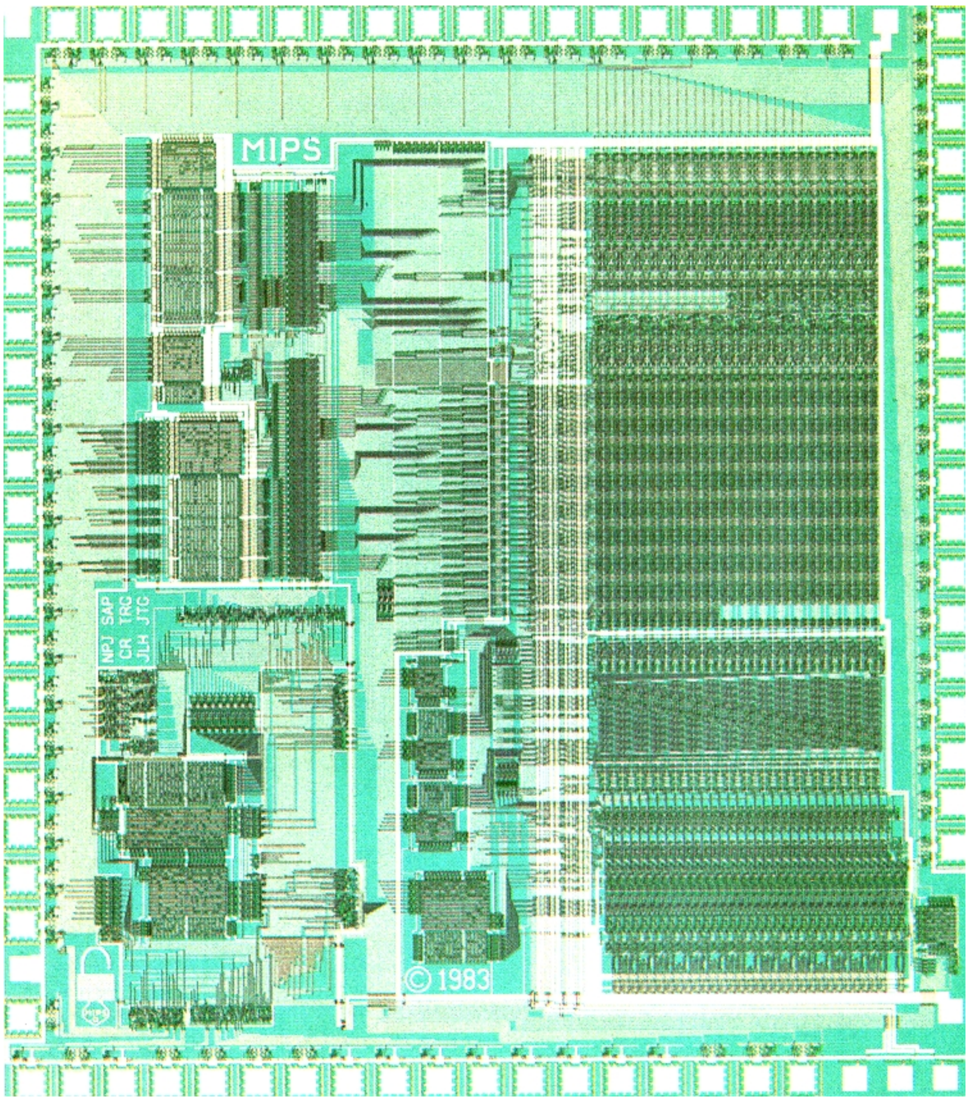
FIGURE 4. Photographs of the Hardware for Three RISC Machines

The Berkeley team did not include compiler experts, so a hardware solution was implemented to keep operands in registers. The first step was to have enough registers to keep all the local scalar variables and all the parameters of the current procedure in registers. Attention was directed to these variables because of two very opportune properties: Most procedures only have a few variables (approximately a half-dozen), and these are heavily used (responsible for one-half to two-thirds of all dynamically executed references to operands). Normally, it slows procedure calls when there are a great many registers. The solution was to have many sets, or windows, of registers, so that registers would not have to be saved on every procedure call and restored on every return. A procedure call automatically switches the processor to use a fresh set of registers. Such buffering can only work if programs naturally behave in a way that matches the buffer. Caches work because programs do not normally wander randomly about the address space. Similarly, through experimentation, a locality of procedure nesting was found; programs rarely execute a long uninterrupted sequence of calls followed by a long uninterrupted sequence of returns. Figure 7 illustrates this nesting behavior. To further improve performance, the Berkeley RISC machines have a unique way of speeding up procedure calls. Rather than copy the parameters from one window to another on each call, windows are overlapped so that some registers are simultaneously part of two windows. By putting parameters into the overlapping registers, operands are passed automatically.

What is the "best" way to keep operands in registers? The disadvantages of register windows are that they use more chip area and slow the basic clock cycle. This is due to the capacitive loading of the longer bus, and although context switches rarely occur—about 100 to 1000 procedure calls for every switch—they require that two to three times as many registers be saved, on average. The only drawbacks of the optimizing compiler are that it is about half the speed of a simple compiler and ignores the register-saving penalty of procedure calls. Both the 801 and the MIPS mitigate this call cost by expanding some procedures in-line, although this means these machines cannot provide separate compilation of those procedures. If compiler technology can reduce the number of LOADs and STOREs to the extent that register windows can, the optimizing compiler will stand in clear superiority. About 30 percent of the 801 instructions are LOAD or STORE when large programs are run; the MIPS has 16 registers compared to 32 for the 801, about 35 percent of them being LOAD or STORE instructions. For the Berkeley RISC machines, this percentage drops to about 15 percent, including the LOADs and STOREs used to save and restore registers when the register-window buffer overflows.

Delayed Loads and Multiple Memory and Register Ports
Since it takes one cycle to calculate the address and one cycle to access memory, the straightforward way to implement LOADs and STOREs is to take two cycles. This is what we did in the Berkeley RISC architecture. To reduce the costs of memory accesses, both the 801 and the MIPS provide "one-cycle" LOADs by following the style of the delayed branch. The first step is to have two ports to memory, one for instructions and one for data, plus a second write port to the registers. Since the address must still be calculated in the first cycle and the operand must still be fetched during the second cycle, the data are not available until the third cycle.





Therefore, the instruction executed following the one-cycle LOAD must not rely on the operand coming from memory. The 801 and the MIPS solve this problem with a *delayed load*, which is analogous to the delayed branch described above. The two compilers are able to put an independent instruction in the extra slot about 90 percent of the time. Since the Berkeley RISC executes many fewer LOADs, we decided to bypass the extra expense of an extra memory port and an extra register write port. Once again, depending on goals and implementation technology, either approach can be justified.

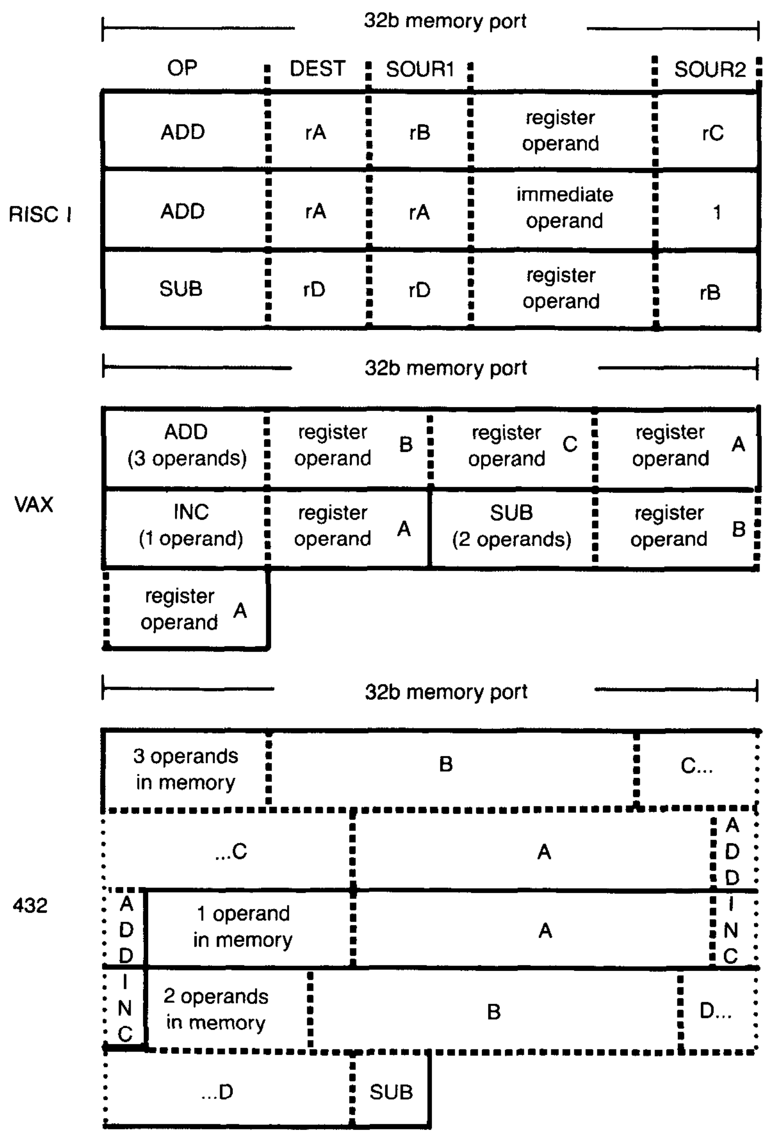
Pipelines

All RISCs use pipelined execution, but the length of the pipeline and the approach to removing pipeline bubbles vary. Since the peak pipelined execution rate is determined by the longest piece of the pipeline, the trick is to find a balance between the four parts of a RISC

TABLE II. Primary Characteristics of Three Operational RISC Machines

	IBM 801	RISC I	MIPS
Year	1980	1982	1983
Number of instructions	120	39	55
Control memory size	0	0	0
Instruction sizes (bits)	32	32	32
Technology	ECL MSI	NMOS VLSI	NMOS VLSI
Execution model	reg-reg	reg-reg	reg-reg

None of these machines use microprogramming; all three use 32-bit instructions and follow the register-to-register execution model. Note that the number of instructions in each of these machines is significantly lower than for those in Table I.



Although variable-sized instructions improve the architectural metrics in Figure 1, they also make instruction decoding more expensive and thus may not be good predictors of performance. Three machines—the RISC I, the VAX, and the 432—are compared for the instruction sequence A ← B + C; A ← A + 1; D ← D − B. VAX instructions are byte variable from 16 to 456 bits, with an average size of 30 bits. Operand locations are not part of the main opcode but are spread throughout the instruction. The 432 has bit-variable instructions that range from 6 to 321 bits. The 432 also has multipart opcodes: The first part gives the number of operands and their location, and the second part gives the operation. The 432 has no registers, so all operands must be kept in memory. The specifier of the operand can appear anywhere in a 32-bit instruction word in the VAX or the 432. The RISC I instructions are always 32 bits long, they always have three operands, and these operands are always specified in the same place in the instruction. This allows overlap of instruction decoding with fetching the operand. This technique has the added benefit of removing a stage from the execution pipeline.

FIGURE 5. Three Machines Are Compared for the Instruction Sequence A ← B + C; A ← A + 1; D ← D − B

instruction execution:

1. instruction fetch,
2. register read,
3. arithmetic/logic operation, and
4. register write.

The 801 assumes that each part takes the same amount of time, and thus uses a four-stage pipeline. We at Berkeley assumed that instruction fetch was equal to the sum of register read and the arithmetic/logic operation, and thus selected the three-stage pipeline shown in Figure 8.

The biggest difference is in handling instruction sequences that cause bubbles in a pipeline. For example, the first instruction in Figure 2b stores a result in A, which the following instruction needs to read. In most machines this forces the second instruction to delay execution until the first one completes storing its result. The 801 and the RISC II avoid inserting a bubble by means of an internal forwarding technique that checks operands and automatically passes the result of one instruction to the next.

The MIPS, in contrast, uses software to prevent interlocks from occurring; this, in fact, is what prompted the machine's name: *M*icroprocessor without *I*nterlocked *P*ipelined *S*tages. A routine passes over the output of the assembler to ensure that there cannot be conflicts. NO-OPs are inserted when necessary, and the optimizing compiler tries to shuffle instructions to avoid executing them. Traditional pipelined machines can spend

TABLE III. A Comparison of the Traditional Branch Instruction with the Delayed Branch Found in RISC Machines

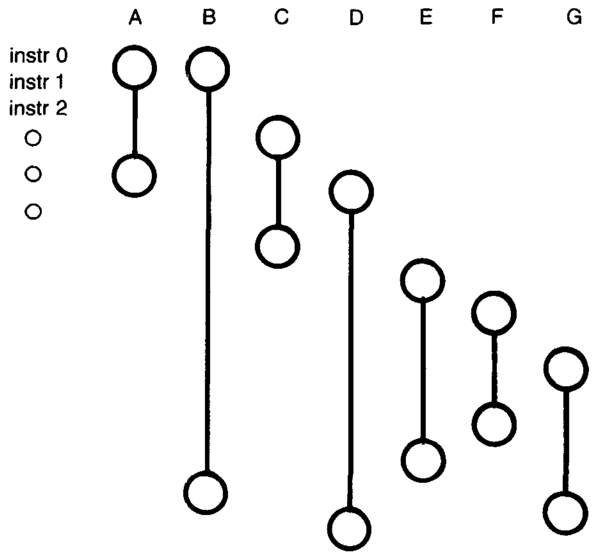
Address	Normal branch		Delayed branch		Optimized delayed branch	
100	LOAD	X,A	LOAD	X,A	LOAD	X,A
101	ADD	1,A	ADD	1,A	**JUMP**	**105**
102	**JUMP**	**105**	**JUMP**	**106**	**ADD**	**1,A**
103	ADD	A,B	**NO-OP**		ADD	A,B
104	SUB	C,B	ADD	A,B	SUB	C,B
105	**STORE**	**A,Z**	SUB	C,B	**STORE**	**A,Z**
106			**STORE**	**A,Z**		

The delayed branch is used to avoid a branch dependency bubble in the pipeline (Figure 2b). Machines with normal branches would execute the sequence 100, 101, 102, 105,.... To get that same effect with a RISC computer, it would be necessary to insert a NO-OP in the Delayed branch column. The sequence of instructions for RISCs would then be 100, 101, 102, *103*, 106,.... In the worst case, every branch would take two instructions. The RISC compilers, however, include optimizers that try to rearrange the sequence of instructions to do the equivalent operations while making use of the instruction slot where the NO-OP appears. The Optimized delayed branch column shows that the optimized RISC sequence is 100, 101, *102*, 105,.... Because the instruction following a branch is always executed and the branch at 101 is not dependent on the add at 102 (in this example), this sequence is equivalent to the original program segment in the Normal branch column.

a fair amount of their clock cycle detecting and blocking interlocks, but the simple pipeline and register model of RISCs also simplifies interlocking. Hennessy believes the MIPS cycle would be lengthened by 10 percent if hardware interlocking were added. The designers of the RISC II measured the internal forwarding logic and found that careful hardware design prevented forwarding from lengthening the RISC II clock cycle.

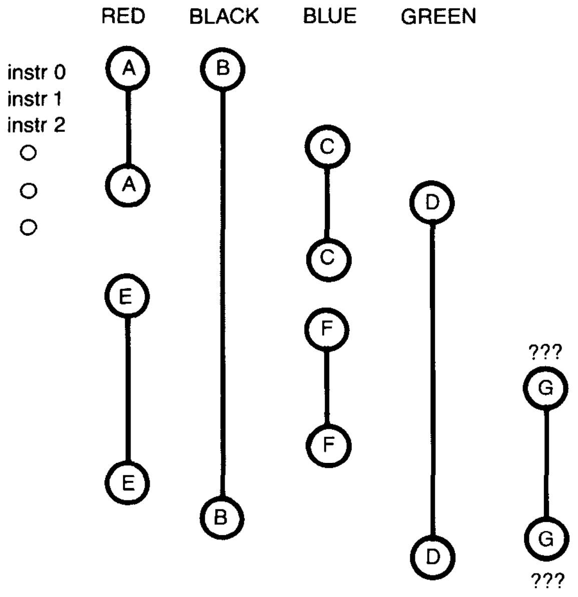
Multiple Instructions per Word

The designers of the MIPS tried an interesting variation on standard RISC philosophy by packing two instructions into every 32-bit word whenever possible. This improvement could potentially double performance if



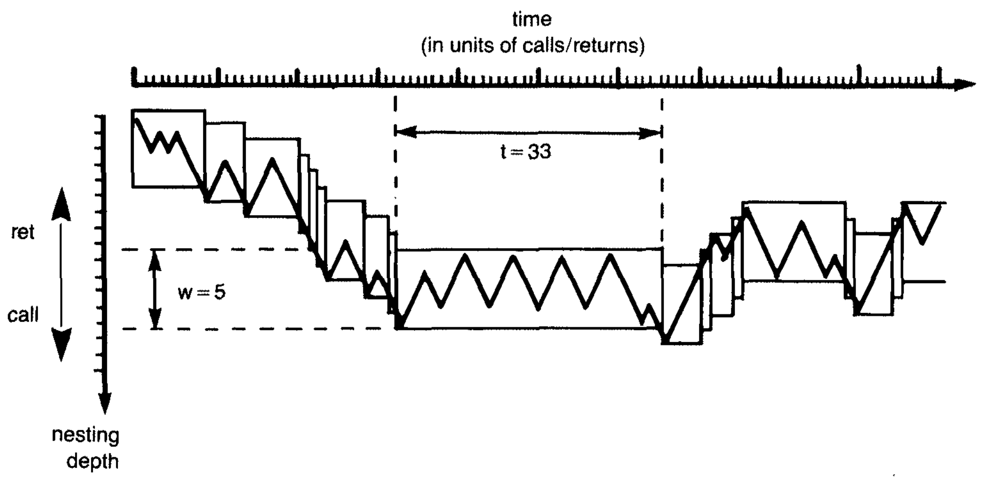
IBM's solution was to deal with register allocation according to the rules prescribed for painting a directed graph with a fixed number of colors. Each color represents one of the machine registers. The simultaneous lifetimes of A through G are shown, from first use to last. Here we assume only four registers, so the problem is to map the seven variables into the four registers. This is equivalent to painting the graph with only four colors.

FIGURE 6a. IBM's Solution to the Register Allocation Problem



We start by mapping the first four variables onto the four colors. Variable E does not conflict with variable A, so they can share the same color, in this case RED. Similarly, F can share BLUE with C. With G all four colors are used, so the compiler must use LOADs and STOREs to free a register to make space for G.

FIGURE 6b. A Partial Solution to Register Allocation



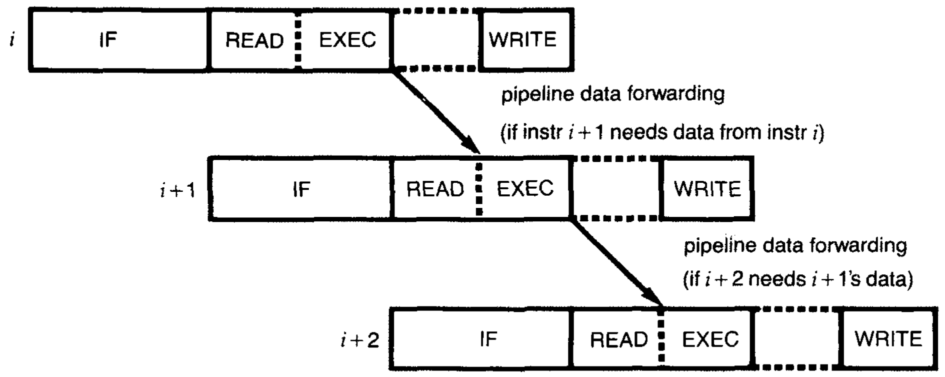
This graph shows the call–return behavior of programs that inspired the register window scheme used by the Berkeley RISC machines. Each call causes the line to move down and to the right, and each return causes the line to move up and to the right. The rectangles show how long the machine stays within the buffer. The longest case is 33 calls and returns ($t = 33$) inside the buffer. In this figure we assume that there are five windows ($w = 5$). We have found that about eight windows hits the knee of the curve and that locality of nesting is found in programs written in C, Pascal, and Smalltalk.

FIGURE 7. The Call-Return Behavior of Programs

twice as many operations were executed for every 32-bit instruction fetch. Since memory-access instructions and jump instructions generally need the full 32-bit word, and since data dependencies prevented some combinations, most programs were sped up by 10 to 15 percent Arithmetic routines written in assembly language had much higher savings. Hennessy believes the two-instruction-per-word format adds 10 percent to the MIPS cycle time because of the more complicated decoding. He does not plan to use the two-instruction-per-word technique in his future designs.

HIDDEN RISCS

If building a new computer around a reduced instruction set results in a machine with a better price/performance ratio, what would happen if the same techniques were used on the RISC subset of a traditional machine? This interesting question has been explored



The memory is kept busy 100 percent of the time, the register file is reading or writing 100 percent of the time, and the execution unit (ALU) is busy 50 percent of the time. The short pipeline and pipeline data forwarding allow the RISC II to avoid pipeline bubbles when data dependencies like those shown in Figure 2b are present.

FIGURE 8. The Three-Stage Pipeline of the RISC II

by RISC advocates and, perhaps unintentionally, by the designers of more traditional machines.

DEC reported a subsetting experiment on two implementations of the VAX architecture in VLSI. The VLSI VAX has nine custom VLSI chips and implements the complete VAX-11 instruction set. DEC found that 20.0 percent of the instructions are responsible for 60.0 percent of the microcode and yet are only 0.2 percent of all instructions executed. By trapping to software to execute these instructions, the MicroVAX 32 was able to fit the subset architecture into only one chip, with an optional floating-point coprocessor in another chip. As shown in Table IV, the VLSI VAX, a VLSI implementation of the full VAX architecture, uses five to ten times the resources of the MicroVAX 32 to implement the full instruction set, yet is only 20 percent faster.

Michael L. Powell of DEC obtained improved performance by subsetting the VAX architecture from the software side. His experimental DEC Modula-2 compiler generates code that is comparable in speed to the best compilers for the VAX, using only a subset of the addressing modes and instructions. This gain is in part due to the fact that optimization reduces the use of complicated modes. Often only a subset of the functions performed by a single complicated instruction is needed. The VAX has an elaborate CALL instruction, generated by most VAX compilers, that saves registers on procedure entry. By replacing the CALL instruction with a sequence of simple instructions that do only what is necessary, Powell was able to improve performance by 20 percent.

The IBM S/360 and S/370 were also targets of RISC studies. The 360 model 44 can be considered an ancestor of the MicroVAX 32, since it implements only a subset of the 360 architecture in hardware. The rest of the instructions are implemented by software. The 360/44 had a significantly better cost/performance ratio than its nearest neighbors. IBM researchers performed a software experiment by retargetting their highly optimizing PL/8 compiler away from the 801 to the System/370. The optimizer treated the 370 as a register-to-register machine to increase the effectiveness of register allocation. This subset of the 370 ran programs 50 percent faster than the previous best optimizing compiler that used the full 370 instruction set.

Software and hardware experiments on subsets of the VAX and IBM 360/370, then, seem to support the RISC arguments.

TABLE IV. Two VLSI Implementations of the VAX

	VLSI VAX	MicroVAX 32	
VLSI Chips (including floating point)	9	2	(22%)
Microcode	480K	64K	(13%)
Transistors	1250K	101K	(8%)

These two implementations, although not yet in products, illustrate the difficulty of building the complete VAX architecture in VLSI. The VLSI VAX is 20 percent faster than the MicroVAX 32 but uses five to ten times the resources for the processor. Both are implemented from the same three-micron double-level metal NMOS technology. (The VLSI VAX also has external data and address caches not counted in this table.)

ARCHITECTURAL HERITAGE

All RISC machines borrowed good ideas from old machines, and we hereby pay our respects to a long line of architectural ancestors. In 1946, before the first digital computer was operational, von Neumann wrote

> The really decisive considerations from the present point of view, in selecting a code [instruction set], are more of a practical nature: the simplicity of the equipment demanded by the code, and the clarity of its application to the actually important problems together with the speed of its handling of those problems.[2]

For the last 25 years Seymour Cray has been quietly designing register-based computers that rely on LOADs and STOREs while using pipelined execution. James Thornton, one of his colleagues on the CDC-6600, wrote in 1963

> The simplicity of *all* instructions allows quick and simple evaluation of status to begin execution. . . . Adding complication to a special operation, therefore, degrades all the others.

and also

> In my mind, the greatest potential for improvement is with the internal methods. . .at the risk of loss of fringe operations. The work to be done is really engineering work, pure and simple. As a matter of fact, that's what the results should be—pure and simple.[3]

John Cocke developed the idea of pushing compiler technology with fast, simple instructions for text and integer applications. The IBM 801 project, led by George Radin, began experimenting with these ideas in 1975 and produced an operational ECL machine in 1979. The results of this project could not be published, however, until 1982.

In 1980 we started the RISC project at Berkeley. We were inspired by an aversion to the complexity of the VAX and Intel 432, the lack of experimental evidence in architecture research, rumors of the 801, and the desire to build a VLSI machine that minimized design effort while maximizing the cost/performance factor. We combined our research with course work to build the RISC I and RISC II machines.

In 1981 John Hennessy started the MIPS project, which tried to extend the state of the art in compiler optimization techniques, explored pipelined techniques, and used VLSI to build a fast microcomputer.

Both university projects built working chips using much less manpower and time than traditional microprocessors.

[2] Burks, A.W., Goldstine, H.H., and von Neumann, J. Preliminary discussion of the logical design of an electronic computing instrument. Rep. to U.S. Army Ordinance Dept., 1946.

[3] Thornton, J.E. *Considerations in Computer Design—Leading Up to the Control Data 6600.* Control Data Chippewa Laboratory, 1963.

CONCLUSIONS

In my view, the remaining technical difficulty is how to get good floating-point performance from RISCs. The RISCs mentioned here have mediocre floating-point performance without special hardware assists. Another issue is whether RISCs will provide the same advantages in the cost/performance factor for exploratory programming environments such as Lisp and Smalltalk. Initial experiments are promising.

I believe the split between architecture and implementation has caused architects to ignore the implementation consequences of architecture decisions, which has led to "good" architectures with disappointing implementations. Complex architecture requires lengthy development cycles, and long development cycles using implementation technologies that routinely double in speed and capacity can potentially mean producing a computer in an antiquated technology. RISCs reduce this danger.

Simple machines can also dramatically simplify design verification. As things stand now, it can take a year or two to discover all the important design flaws in a mainframe. VLSI technology will soon allow a single-chip computer to be built that is as complicated as a mainframe. Unfortunately, VLSI manufacturing will also produce hundreds of thousands of computers in the time it takes to debug a mainframe in the field. RISC machines offer the best chance of heading off "computer recall" as a new step in the development cycle.

A final observation: Technology runs in cycles, so trade-offs in instruction set design will change over time, if there is change in the balance of speeds in memory and logic. Designers who reconcile architecture with implementation will reduce their RISCs.

Acknowledgments. I thank the many Berkeley students who worked hard to create RISCs, and my colleagues at IBM and Stanford for pursuing RISCs. I must also acknowledge the excellent papers by Hennessy and Hopkins for providing me with insights to help explain the RISC phenomenon.

Thanks also go to Gordon Bell, Peter Denning, Susan Eggers, Robert Garner, Garth Gibson, Paul Hansen, John Hennessy, Paul Hilfiger, Mark Hill, Normam Jouppi, Manolis Katevenis, Randy Katz, John Ousterhout, Joan Pendleton, Michael Powell, George Taylor, Tom West, and David Ungar for their useful suggestions in improving this paper.

The Berkeley RISC project was sponsored by DARPA, order number 3803, monitored by NAVALEX under contract number N00034-K-025.

Further Reading. Items [1, 4, 9] should provide a good introduction to RISCs for the general reader. The remaining references may help to explain some of the ideas presented in this paper in greater depth. The IBM 801 is described in [12], while the two papers by Martin Hopkins of IBM [5, 6] provide historical and philosophical perspective on RISC machines. The IBM graph-coloring algorithm is found in [2], and the subsetted use of one instruction set in [8]. The primary reference for the Berkeley RISC machines is [11], although [10] contains the latest information on performance of the RISC II and [7] a thorough explanation of the motivation for RISCs in general and the RISC II in particular. An attempt to use RISCs in an exploratory programming environment, in this case the object-oriented Smalltalk system, is described in [13]. Joseph Fisher, a Writable Control Store refugee like myself, is now working on compiling "normal" programming languages into very wide instructions (or horizontal microinstructions, depending on your perspective) for a high-performance multiple arithmetic unit computer [3].

REFERENCES

1. Bernhard, R. More hardware means less software. *IEEE Spectrum 18*, 12 (Dec. 1981), 30–37.
2. Chaitin, G.J. Register allocation and spilling via graph coloring. In Proceedings of the SIGPLAN 82 Symposium on Compiler Construction. *SIGPLAN Not.* 17, 6 (June 1982), 98–105.
3. Fisher, J.A. Very long instruction word architectures and the ELI-512. In *The 10th Annual International Symposium on Computer Architecture* (Stockholm, Sweden, June 13–17). ACM, New York, 1983, pp. 140–150. ←
4. Hennessy, J.L. VLSI processor architecture. *IEEE Trans. Comput.* To be published.
5. Hopkins, M. A perspective on microcode. In *Proceedings of the 21st Annual IEEE Computer Conference (Spring COMPCON 83)* (San Francisco, Calif., Feb.). IEEE, New York, 1983, pp. 108–110.
6. Hopkins, M. Definition of RISC. In *Proceedings of the Conference on High Level Language Computer Architecture* (Los Angeles, Calif., May). 1984.
7. Katevenis, M.G.H. Reduced instruction set computer architectures for VLSI. Ph.D. dissertation, Computer Science Dept., Univ. of California, Berkeley, Oct. 1983.
8. Lunde, A. Empirical evaluation of some features of instruction set processor architecture. *Commun. ACM 20*, 3 (Mar. 1977), 143–153.
9. Patterson, D.A. Microprogramming. *Sci. Am. 248*, 3 (Mar. 1983), 36–43.
10. Patterson, D.A. RISC watch. *Comput. Archit. News 12*, 1 (Mar. 1984), 11–19.
11. Patterson, D.A., and Séquin, C. A VLSI RISC. *Computer 15*, 9 (Sept. 1982), 8–21.
12. Radin, G. The 801 minicomputer. *IBM J. Res. Dev. 27*, 3 (May 1983), 237–246.
13. Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Symposium on Computer Architecture* (Ann Arbor, Mich., June 5–7). ACM, New York, 1984, pp. 188–197. ←

CR Categories and Subject Descriptors: B.1.1 [**Control Structures and Microprogramming**]: Control Design Styles; B.2.1 [**Arithmetic and Logic Structures**]: Design Styles; B.7.1 [**Integrated Circuits**]: Types and Design Styles; C.1.0 [**Processor Architectures**]: General
General Terms: Design, Performance
Additional Key Words and Phrases: RISC, VLSI, microprocessors, CPU

Author's Present Address: David A. Patterson, Dept. of Electrical Engineering and Computer Sciences, Computer Science Division, University of California, Berkeley, CA 94720.