The evolution of RISC technology at IBM

by John Cocke
V. Markstein

This paper traces the evolution of IBM RISC architecture from its origins in the 1970s at the IBM Thomas J. Watson Research Center to the present-day IBM RISC System/6000* computer. The acronym RISC, for Reduced Instruction-Set Computer, is used in this paper to describe the 801 and subsequent architectures. However, RISC in this context does not strictly imply a reduced number of instructions, but rather a set of primitives carefully chosen to exploit the fastest component of the storage hierarchy and provide instructions that can be generated easily by compilers. We describe how these goals were embodied in the 801 architecture and how they have since evolved on the basis of experience and new technologies. The effect of this evolution is illustrated with the results of several benchmark tests of CPU performance.

Introduction

IBM RISC technology originated in 1974 in a project to design a large telephone-switching network capable of handling an average of three hundred calls per second. With an approximate 20 000 instructions per call and stringent real-time response requirements, the performance target was 12 million instructions per second (MIPS) [1]. This specialized application required a very fast processor, but did not have to perform complicated instructions and had little demand for floating-point calculations. Other than moving data between registers and memory, the machine had to be able to add, combine fields extracted from several registers, perform branches, and carry out input/output operations.

When the telephone project was terminated in 1975, the machine itself had not been built, but the design had progressed to the point where it seemed to be an excellent basis for a general-purpose, high-performance miniprocessor. The attractiveness of the processor design stemmed from projections that it would be able to compute at high speed relative to its cost in a variety of application areas.

The most important features of the telephone-switching machine which contributed to its low cost/performance ratio were 1) separate instruction and data caches, allowing a much higher bandwidth between memory and CPU; 2) no arithmetic operations to storage, which greatly simplified the pipeline; and 3) uniform instruction length and simplicity of design, making possible a very short cycle time: ten levels of logic. (For example, all register-to-register operations executed in one cycle.)

Instruction traces showed that 30 percent of all instructions involved moving data between storage and the CPU [2]. Various studies had shown that branching can take up as much as one third of the execution time [2]. Performance degradation due to branches had been recognized as far back as the IBM 7030 (STRETCH), where the hardware was biased to perform better if

*RISC System/6000 is a trademark of International Business Machines Corporation.

REPRINTED FROM IBM JOURNAL OF RESEARCH AND DEVELOPMENT, VOL. 34, NO. 1, 1990; ©1990, 2000

0018-8646 / 00 / $5.00 © 2000 IBM

conditional branches failed to execute the target instruction.

A small degree of pipelining, two or three levels, cut down the effective time required for memory accesses and branches. Memory-fetch operations required two cycles, one to compute the address of the data and send the address on the memory bus, and a second to receive the data and place it in the target register. Since the CPU itself was not needed during the second cycle, it was available to execute the following instruction unless that instruction required the data being fetched.

Pipelining in instruction fetching also helped reduce the cost of branching, which required one or two cycles depending on whether the branch was successful. During the first cycle, the target address was computed, and it was determined whether the branch would be successful. If the branch was unsuccessful, the machine would continue with the prefetched instruction in the current instruction stream. If the branch was successful, however, a cycle would be lost while the first word of the new instruction stream was fetched from memory and reached the CPU. To recover this cycle, a second form of branch instruction, called BRANCH AND EXECUTE, was introduced in an experimental successor to the telephone-machine design. This form of branch, which is commonly called "delayed branch," caused the CPU to unconditionally execute the instruction immediately following the branch, whether or not the branch was successful. If a BRANCH AND EXECUTE could be used instead of a conventional branch instruction, the lost cycle could be recovered whenever the branch was taken. Delayed branch, by the way, was used as early as 1952 in the Los Alamos MANIAC computer [3] and was resurrected in the experimental machine.

To keep the basic cycle as short as possible, the original machine was architected without the usual memory-protect mechanisms, which require memory blocks to have control bits to indicate whether they are read, write, or execution blocks. The machine would have depended entirely on software for this feature, but this notion was abandoned in later designs. The cache, a high-speed memory used as a buffer between main memory and the CPU, was split into two parts: a data cache and an instruction cache. At that time it was widely accepted that a running program would not modify itself at execution time. Therefore, no mechanisms were added to ensure that stores into the instruction stream were immediately reflected in the instruction cache. Instead, the ability to void cache lines was added to the instruction set.

When the capabilities of this machine were compared with those of large System/370 machines, it was realized that its programs would be longer: Many of the System/370 instructions which required many cycles were no longer available. Even the popular ADD FROM STORAGE instruction was gone, and the equivalent operation would require a LOAD instruction followed by an ADD FROM REGISTER instruction. However, the System/370 Model 168 (for example) took almost as long to execute the ADD FROM STORAGE instruction as the two-instruction sequence that our experimental machine would require, and the latter was in fact an alternative sequence for the System/370.

IBM had a vast amount of data on instruction frequencies for various applications [1]. From these instruction traces, it was clear that LOAD, STORE, BRANCH, FIXED-POINT ADD, and FIXED-POINT COMPARE were the most frequently occurring instructions, accounting for well over half of the total execution time in most application areas. (Numerically intensive computation was the one exception, where floating-point operations were among the instructions most frequently seen.) Therefore, the experimental machine did not seem to be disadvantaged, since it could execute each of the most often used System/370 operations in one cycle. In that sense, the machine was very similar to a vertical microcode engine, i.e., a machine that executes one instruction at a time. But instead of "hiding" this attribute behind a complex instruction set in microcode, we exposed it directly to the end user. Removing that level of indirection allowed the most frequent instructions to be executed in one machine cycle each, whereas using the machine as a microcomputer simulating a System/370 would introduce a simulator overhead of about ten instructions. Only after that would the machine as a simulator issue the one-instruction equivalent of the fastest System/370 instructions.

This was the key realization: Imposing microcode between a computer and its users imposes an expensive overhead in performing the most frequently executed instructions. Thus, a key task in designing the experimental machine was to investigate the consequences of exposing a microcomputer directly to the end user. In many cases, a microcomputer limited to instructions executable in one cycle would execute a macro-instruction in about as many cycles as a System/370 Model 168 executing the equivalent instruction. The great potential was that simple instructions would run substantially faster for the same circuit family and cycle time because the overhead of executing a CISC (Complex Instruction-Set Computer) interpreter was pared away.

Of course, at that time, the acronym "CISC" did not yet exist. Neither did "RISC," and for a time there was no name for the experimental computer. "The telephone machine" began to seem inappropriate, and we named the machine "the 801" after the designation of the IBM building in which the research was taking place.

Interaction of software with the 801

Compilers were expected to play a central role in the 801. Its architecture was the antithesis of the "semantic gap" idea, in that instructions were specifically designed for efficient use by a compiler. Because of the lack of hardware memory protection, it was envisioned that every piece of code would be written in high-level languages for which compilers could be provided. The compilers ensured that code would not make accesses outside the regions in which it was entitled to operate.

For memory protection, a special instruction, the trap instruction, was introduced into the 801. This instruction compared two quantities, and if a specified condition existed between the two quantities, the 801 "trapped" (took its next instruction) from a fixed, hardware-determined location. The trap instructions behaved like sequential instructions, which on rare occasions might trap (just as a division on rare occasions would cause a divide check). The compilers were to compare memory addresses with bounds, and cause traps if the accesses were outside the bounds. Of course, the compilers were also expected to "optimize away" most of the traps and the instructions which triggered them [4].

In mid-1975, there existed no software at all for the 801. An assembler was quickly fashioned, and a simulator was designed and built. The simulator was especially fast, and its design was strongly influenced by the split-cache design of the 801. Since the instruction cache of the 801 was not expected to reflect changes to memory unless explicitly synchronized, the simulator was designed to simply translate each 801 instruction into 32 bytes of System/370 code which would implement the 801 instruction. Every time a new cache line was accessed, the 801 memory would be compiled into System/370 code, and only on cache-invalidate instructions (801 instructions which specifically indicate that specified portions of the cache are no longer valid) would the translated code be abandoned. The result was a simulator that ran at about one-tenth the speed of the host machine, fast enough to simulate meaningful programs.

There was still the question of higher-level languages. With limited resources, it was decided to concentrate on just one. PL/I seemed to be a desirable language, since it supported many applications. Its very richness, however, also made it difficult for compilers to produce good code over much of its capability. The construct of PL/I was therefore reduced to a subset useful to system programmers, and those language features which seemed to defy reasonable translation were discarded. The more arcane constructs would be coded by the programmer using the rational subset of PL/I that was recognized. The result was the PL.8 language [5], the ".8" implying that it had about 80 percent of the richness of PL/I. PL.8 bore the same relation to PL/I as the 801 architecture had to the System/370.

Initially, PL.8 was a pure subset of PL/I, so that the compiler could be coded in PL/I, developed on a System/370, and its output (801 code) tested on the simulator. Because there was great concern initially that 801 code sequences would be long and cumbersome, compiler-code quality was always a central objective in the construction of the PL.8 language and its compilers. To that end, the general optimization algorithms described in [6] were used. (Many of these algorithms, while very general and powerful, had not previously been used outside the classroom.)

Our approach to register allocation, which was deemed to be central to the proper use of the 801, was "graph coloring." This approach had been mentioned in the literature [7], but was implemented for the first time in the PL.8 compiler [8].

PL.8 compiler output is code which can be executed on the 801. Using the PL.8 compiler to compile itself on a System/370 produced a PL.8 compiler that would execute on an 801 machine. (This method of producing a compiler for a new architecture is called "bootstrapping.")

As the compiler grew in its capabilities, we were able to simulate sizable pieces of 801 code. Eventually we bootstrapped the compiler and ran it on the simulator as well. The simulator counted the number of instructions executed, and from this number we could project how fast a program could run on the real 801, once the engineers finished constructing a model. Many of the results were very favorable to the 801 architecture; ironically, however, the compiler proved to be a bad case! The culprit was the large number of move-strings occurring in the compiler, because the 801 lacked a sufficiently powerful means of moving strings that were not identically aligned with respect to full-word boundaries.

However, one interesting outcome of the bootstrap procedure was the discovery of dozens of uses for uninitialized data. PL/I failed to pick these up because the cost of executing PL/I code compiled with SUBSCRIPTRANGE ON and other checking code enabled was intolerably high. PL.8, which always ran with all checking enabled, but whose compiler optimized away the vast majority of the overhead, discovered these uses for uninitialized data because, when used later as subscripts, this data would have inappropriate values.

As the optimizer and the register-allocation techniques [9] improved, it was discovered that the resultant 801 code was not much different from ordinary System/370 code. The code sequences were not unduly long or unnatural. (In later years, path-length comparisons between RISC and CISC architectures have been shown

VOL. 34, NO. 1, 1990, REPRINT

to be very nearly equal [10].) The path-length result was mostly due to the ability of the compiler to perform greater optimization, which in turn was possible because of the regular instruction format. (Had we decided to name our architecture at that time, we might have called it a Regular Instruction-Set Computer.)

On the whole, the code generated for the 801 confirmed our belief that an exposed vertical-microcode machine was a very cost-effective, high-performance machine. (The original 801 instruction set was adopted by the IBM Office Products Division as the basis for a microcomputer, a transfer of technology that in time developed into the IBM RT System.) There were, however, several areas in which the 801 needed improvement. String handling has been already cited above. Decimal arithmetic was found to need hardware that could propagate carries between four-bit subfields. The register allocator proved to be very effective, but it showed that as many as 32 registers would be desirable to take full advantage of the architecture. In addition, register allocation had to cope with the fact that most 801 operations replaced one of the operands with the result, thereby making it costly to reuse that operand; an instruction format which allowed the result register to be specified independently of the input registers could have been used to great advantage by the 801 style of register allocation.

Finally, the original 801 supported a maximum of 16 megabytes of memory. With the radical reduction in the cost of memory that was occurring at the time, it became clear that a competitive RISC computer would require significantly larger addressability, and that virtual memory could not be ignored. To accommodate these requirements, a second 801 design was begun.

Improved 801 architectures

The second 801 reflected the lessons learned from the first. First, all instructions would now be 32 bits in length. This simplified the instruction-decode mechanism and made it easier for look-ahead mechanisms to classify operations. Instructions could no longer straddle cache lines, and with the adoption of virtual addressing, full-word instructions would also never straddle page boundaries. All of these benefits further simplified instruction manipulation, and served to nullify the extra complexities that virtual memory added to instruction handling and memory referencing.

For the original 801, the average instruction length was found to be three bytes. The second 801, with its fixed four-byte instruction length, had its program size increased by less than a factor of 33 percent, because fewer instructions were needed.

The 32-bit instruction was long enough to reference 32 registers, and to provide a unique field to specify the result of the operation. Nondestructive instructions allowed better reuse of data, and the additional 16 registers avoided most of the register-spilling code (code to store and reload registers) which resulted when register allocation failed in its initial attempt with the limitation of only 16 registers. As long as no spill-code was introduced, the compiler could easily outperform hand-coding.

The fixed-point unit was bolstered by a powerful rotator, which was capable of rotating the contents of one register and combining selected bits of the rotated result with the contents of another register, the result being delivered either to a third register or to storage. The ROTATE instructions significantly improved the performance of the 801 move-character routine, and provided powerful operations for the combining of bit fields that occurs often in compilers and operating systems. The more powerful SHIFT instructions also reduced the time needed to simulate floating-point instructions. The SHORT FLOATING ADD, for example, could be performed in 20 cycles.

Assists were provided for decimal addition and subtraction, which allowed two words of eight digits to be added in only four instructions.

A major advance in the 801 CPU was its ability to branch based on the state of any bit in any general-purpose register. As a result, the state of the condition register could be saved in a general-purpose register, improving the treatment of several other branches against the same condition that might occur in widely separated parts of a program. With most other architectures, saving the condition code and then branching on the saved information is so cumbersome that optimization of conditional branches is not feasible. For the new 801, it had become a desirable technique.

In accessing storage, the 801 could add the contents of a base register either to an immediate operand found in the instruction itself, or to the contents of another register. The new 801 included the notion we call progressive indexing, in which the effective address replaces the contents of the base register.

The original compiler was modified to generate code for the improved architecture. Reassociation [11] was added to the collection of optimizations to improve addressing in loops. This new optimizing technique is an extension of strength reduction, which exploits the associative law of addition to expose additional common subexpressions that can later be moved out of loops or be discarded as redundant computations. Reassociation ultimately enabled progressive indexing to be used in many commonly occurring loops.

Better and faster spill heuristics enhanced the register allocator, which was easily parameterized to handle the 32 registers of the 801 or the 16 of System/370. Of

course, spilling was a much rarer event on the enhanced 801. Removing accesses to main memory continued to be pervasive in the compiler.

System/370 as a RISC machine

The 801 team was pleased with the results of its labors, even though by 1977 it had only simulator results on which to rely. Compiler code quality was high; when compared with code produced by the PL/I compiler, the contrast was impressive. There even were instances of code which, when compiled for the 801 and simulated on a System/370 Model 168, ran faster in real time than the same program run directly on a Model 168 when compiled by the PL/I compiler.

When a System/370 processor was added to the compiler, little was done to alter the code-generation patterns from those used on the 801. As a consequence, System/370 RX instructions such as add-from-memory-to-register went unused, being replaced by the 801 version 2 instruction sequence (load-from-memory, register-to-register-add). Of course, the System/370 suffered from register-operand destruction, as did the original 801.

Once PL.8 was ported to System/370, it no longer had to be a strict subset of PL/I. Perhaps the most RISC-like change to the language was to enable the programmer to specify that arguments were to be passed by value. This enabled the argument itself to be passed in a register in keeping with the general 801 viewpoint, and also freed the optimizer from having to make pessimistic assumptions about the use of arguments by subprograms.

We discovered that our Model 168, running code generated by the PL.8 compiler, consistently ran between 4.5 and 6 MIPS at a time when it was considered an accomplishment to drive the 168 at 2 MIPS. Perhaps the path-lengths were somewhat longer, but certainly not 50 percent longer. Abstaining from the CISC-like operations of the System/370 and using it as a RISC machine gained substantial performance improvement. This was largely due to the effort of the PL.8 compiler to reuse data already present in the registers of the System/370. Also contributing to the performance of PL.8 code on the System/370 were the streamlined subroutine prologues and epilogues made possible by the PL.8 register conventions and simple run-time environment. This demonstrated conclusively that an appropriate combination of RISC-based architecture and an optimizing compiler can outperform a CISC-based CPU for a comparable program-instruction stream without materially expanding the program code [12].

The payoff of this code-generation technique was, of course, reduced by later, more powerful System/370s, whose more aggressive pipelining and caching drastically reduced the overhead of certain storage accesses. PL.8 adjusted its code-selection techniques over the years to use more of the System/370 CISC instructions, but the emphasis of the software (as well as IBM RISC architecture) continues to be to reuse information in the fastest storage elements (registers) to the greatest degree possible.

801 technology transfer

The original 801 was completed in 1978, and for a time was IBM's fastest experimental processor. In the meantime, several planned IBM development projects used 801s as microcomputers. The IBM 3090 I/O processor uses a 40-MHz 801 as its engine, and a good portion of its code was written in PL.8. The IBM 9370 uses an 801 as its microcomputer. The newer 801 instruction set was also enhanced with several special-purpose instructions to assist in the simulation of System/370.

The Toronto Language Project adopted PL.8 optimization and register-allocation technology for use in the postprocessor of the XL family of retargetable compilers [13]. These compilers can produce code for a wide variety of platforms such as the IBM PS/2 386 models, the IBM RT System, System/370, and the IBM RISC System/6000* computer. (The algorithms have been reimplemented with attention to compiler efficiency, which was a secondary consideration for the 801.)

At the same time, PL.8 was enhanced with a Motorola M68000 "back-end" [11], and was used for a number of products incorporating that microprocessor.

The RISC System/6000 computer

The goal of the 801 family was to execute one instruction per cycle. While this execution rate can be achieved in specialized code, this rate has not been realized in general code. Very-large-scale integration (VLSI), however, has significantly increased circuit density and opened the possibility of using additional pipelining to smooth out delays caused by storage accesses and conditional branching.

To take advantage of this, a new design evolved which provides three semi-autonomous processors: an instruction-stream processor, a fixed-point processor, and a floating-point processor. The new machine has a very fast floating-point multiply-add unit and is capable of concurrently executing a fixed-point, a floating-point, and a branch instruction. Details of this design are found in companion papers [14, 15]. The optimizing compiler was essential to exploit this capability; a description of the compiler and discussions of related instruction-scheduling techniques are provided in [16, 17]. The experimental version of the design, called AMERICA and developed at the Thomas J. Watson Research Center, was subsequently transferred to the development

Table 1 CPU benchmark performance for selected workstations.

CPU	*Dhrystones 1.1*	*Whetstones (millions)*	*Linpack(dp) Fortran rolled*	*Livermore loops (geometric mean)*	*SPEC* Mark*
Sun 4/200	19 000	3.9	1.6	0.77	—
DecStation 3100	25 000	8.8	1.6	1.99	10.1
Apollo DN10000	25 461	14.9	5.1	2.50	13.9
MIPS M/2000	43 100	14.1	3.9	3.60	17.6
RISC System/6000 Processor	60 700	25.5	10.9	8.90	28.9

*SPEC is a trademark of Systems Performance Evaluation Cooperative.

laboratory in Austin, where it evolved into the RISC System/6000 (RS/6000) processor.

From a software viewpoint, the instruction set is still simple. The machine behaves as though it executes straight-line code sequentially; although instructions are actually not executed sequentially (since there are three processors that can operate concurrently), the programmer and the compilers are shielded from the parallel effects. Thus, the RS/6000 processor retains the property of earlier 801s of having an instruction set that is readily usable by the compiler.

The RS/6000 instruction set has been enhanced, however, with some decidedly complex instructions. Most notable is the inclusion of floating-point instructions, most of which operate in two cycles [14, 15]. These were added to the orginal 801 instruction set because floating-point instructions occur frequently in most scientific, engineering, and visualization applications. Therefore, a floating-point RISC architecture supports the notion of an optimized hardware implementation for the most frequently used primitives, which could not be provided as efficiently with vertical microcode; no one-cycle instructions have led to acceptable performance for floating-point computation.

The floating-point unit actually has more real registers than can be addressed by instructions. A register-remapping scheme allows several independent sequences using the same architected register to be processed concurrently in the pipeline. This capability is vital, since the floating-point pipeline can contain all the instructions for several iterations of short loops. In this way, it is not necessary to delay the decoding of later instructions until earlier uses of a register are completed.

String-move and string-compare assists have also been added to the RS/6000 instruction repertoire to reduce the start-up time for unaligned character-move sequences.

The instruction-stream processor contains eight 4-bit condition registers. Experience with older 801s had shown the benefit of moving compares out of loops. In those machines, the contents of a condition register could be copied into a general-purpose register and later tested by the CPU. In the RS/6000 CPU, the general-purpose registers are in a different unit from the instruction processor, and a mechanism for preserving multiple condition-register results without access to the general-purpose registers added hardware to the instruction processor. The instruction-stream processor also has the ability to perform logical operations on bits of the condition registers, thereby relieving the execution units of the task of computing complex branching conditions. The ability to preserve, in a manner convenient to the instruction-stream processor, the results of several comparisons makes branching free when the outcomes of these comparisons must be reexamined.

The new capabilities of the RS/6000 processor require additional compiler techniques to exploit them. The RS/6000 loop-closing instruction is a consequence of reassociation. The capabilities of the branch unit impose new considerations for the compiler's scheduler [16, 17]. Techniques often used for compilers for vector processors are also appropriate for the RS/6000 procressor. These include loop unrolling, loop jamming, and "strip mining."

Not only does the RISC System/6000 CPU have built-in floating-point execution, but it can compute $\pm z \pm xy$ with one instruction, and with only one rounding error. This instruction provides capabilities beyond those required by the IEEE floating-point standard [18], and provides interesting new opportunities for numeric applications [19].

To illustrate the results of the design, **Table 1** shows comparisons between a 25-MHz development model of the RS/6000 CPU and a number of other workstations against some commonly used CPU-performance benchmarks [20].

Summary

Here we recapitulate the design principles, evolved from experience with the family of 801 machines, that are embodied in the RISC System/6000 processor. The principal objective was to realize an architecture that would achieve the performance gains promised by RISC technology while maintaining the efficiency of conditional-branch handling and floating-point operations required of a technical workstation. We have certainly abandoned the one-cycle-per-instruction

concept in those cases where vertical microcode cannot match more complex hardware, as in the case of floating-point arithmetic. From today's vantage point, the architectural aims can be summarized as follows:

- Design instructions to use the fastest portion of the memory hierarchy effectively, and to enable data in the fastest memory to be reused as much as possible. (For the 801 family, the fastest memory is the set of registers.)
- Provide many functionally equivalent copies of the fastest memory. (That is, avoid having a large set of one-of-a-kind objects.)
- Avoid complex instructions *whenever the same effects can be realized just as quickly by sequences of simple instructions*. For a given application, cache will become the equivalent of fast ROM for the macro-instructions most commonly used by that application.
- Use separate instruction and data caches to materially increase the bandwidth of the data path to the backing store, and explicit cache-invalidation instructions to simplify and speed up the instruction-fetching mechanism.
- Ensure that all instructions are usable by compilers.
- Provide an optimizing compiler which can accommodate the architecture's scheduling requirements, and which can effectively use the fastest memory of the machine (i.e., registers).

Companion papers [14–17, 19] describe the completed architecture of the IBM RISC System/6000 processor and the details of hardware design which have realized these objectives.

References

1. John Cocke, "1987 Turing Award Acceptance Speech," *Commun. ACM* **31,** 250 (March 1988).
2. J. C. Gibson, "The Gibson Mix," *Technical Report TR00.2043*, IBM Systems Development Division, Poughkeepsie, NY, 1970.
3. D. R. Ditzel and H. R. McLellan, "Branch Folding in the CRISP Microcomputer: Reducing Branch Delays to Zero," *Proceedings of the 14th Annual Meeting of the International Society for Computing Architecture, Tokyo, 1987,* IEEE, New York, Cat. No. 87CH2420-8 (1987).
4. V. I. Markstein, J. Cocke, and P. W. Markstein, "Optimization of Range Checking," *ACM SIGPLAN Notices* **17,** 114–119 (1982).
5. M. E. Hopkins and M. A. Auslander, "An Overview of the PL.8 Compiler," *ACM SIGPLAN Notices* **17,** 22–31 (1982).
6. J. Cocke and J. Schwartz, "High Level Languages and Their Compilers," *Courant Computer Science Notes* **C66,** Courant Institute of Mathematical Sciences, New York University, New York, 1969.
7. J. T. Schwartz, "On Programming: An Interim Report on the SETL Project," *Courant Computer Science Notes* **S91,** Courant Institute of Mathematical Sciences, New York University, New York, 1973.
8. G. Chaitin, "Register Allocation Via Coloring," *Comput. Lang.* **6,** 47–57 (1981).
9. George Radin, "The 801 Minicomputer," *IBM J. Res. Develop.* **27,** 237–246 (1983).
10. J. Hennessy, "Overview of the Stanford UCode Compiler System," (monograph), Stanford University, Stanford, CA, 1982.
11. J. Cocke and P. W. Markstein, "Measurements of Program Improvement Algorithms," *Proceedings of IFIP 80,* North-Holland Publishing Co., Amsterdam, 1980, pp. 221–228.
12. P. Wallich, "Toward Simpler, Faster Computers," *IEEE Spectrum* **22,** 38–45 (1985).
13. *AIX/RT XL FORTRAN Users Guide*, Order No. SC09-1268, IBM Canada Ltd., 1989; available through IBM branch offices.
14. G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 37–58 (1990, this issue).
15. R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 Floating-Point Execution Unit," *IBM J. Res. Develop.* **34,** 59–70 (1990, this issue).
16. H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 85–92 (1990, this issue).
17. M. C. Golumbic and V. Rainish, "Instruction Scheduling Beyond Basic Blocks," *IBM J. Res. Develop.* **34,** 93–97 (1990, this issue).
18. "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Standard No. 754*, American National Standards Institute, Washington, DC, 1988.
19. P. W. Markstein, "Computation of Elementary Functions on the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 111–119 (1990, this issue).
20. "Performance Brief: CPU Benchmarks," *Performance Brief 3.9*, John Mashee, Ed., MIPS Computer Systems, Inc., Sunnyvale, CA, January 1990. A description of the Dhrystone and LINPACK benchmarks and the conditions of their application to the RISC System/6000 processor is given in H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye, "The IBM RISC System/6000 Processor: Hardware Overview," *IBM J. Res. Devel.* **34,** 12–22 (1990, this issue).

Received March 22, 1989; accepted for publication December 18, 1989

John Cocke *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Cocke received his B.S. in mechanical engineering from Duke University, Durham, North Carolina, in 1946, and his Ph.D. in mathematics, also from Duke University, in 1956. He joined the IBM Research Division the same year. Dr. Cocke was appointed an IBM Fellow in 1972; with an IBM colleague, he received the ACM Programming Systems and Languages Award in 1976. He was elected to the National Academy of Engineering in 1979, received the ACM/IEEE Computer Society Eckert–Mauchly Award in 1985 and the ACM A. M. Turing Award in 1987; he became a Fellow of the American Academy of Arts and Sciences in 1988. Dr. Cocke has been a visiting professor at the Massachusetts Institute of Technology and at New York University's Courant Institute of Mathematical Sciences. His major area of research interest continues to be systems architecture, particularly hardware design and program optimization.

Victoria Markstein *IBM University and College Systems, 1000 Westchester Avenue, White Plains, New York 10601.* Mrs. Markstein received a B.S. in physics from Queens College, New York, and an M.S. in computer sciences in 1967 from Pratt Institute, Brooklyn, New York. She was also a graduate student at New York University, Courant Institute of Mathematical Sciences. Mrs. Markstein was a faculty member of the Mathematics Department at Lehman College from 1970 to 1975; she was on the adjunct Computer Science faculty of Pratt Institute and New York University. From 1978 to 1988, she was a Research Staff member at the IBM Thomas J. Watson Research Center, where she worked with John Cocke on compiler optimizations and architecture design of RISC-like machines. She also worked as technical staff to the Vice President for Advanced Engineering, Entry Systems Division, from 1988 to 1989. Mrs. Markstein holds several patents in the area of compiler optimization. She is currently program manager of Computer Sciences for University and College Systems, Entry Systems Division.