Feature

Just Curious: An Interview with John Cocke

This computer architecture leader's curiosity led him to discover several of the field's most significant advances.

Bruce Shriver
Genesis 2 Inc.

Peter Capek
IBM T.J. Watson Research Center

A legend in the computer architecture community, John Cocke has been involved in the design of several machines that have made a tremendous impact on current processor design, including the IBM Stretch; the Advanced Computer System (ACS); and the 801, RS/6000, and PowerPC processors.

Perhaps best known as a pioneer of ideas that led to reduced instruction set computing (RISC), Cocke is also much admired for a broad interest in and understanding of technology that spans mathematics, compilers, architecture, circuits, packaging, and design automation, to name a few. In conjunction with his winning the inaugural Seymour Cray Award, *Computer* visited Cocke in his Westchester, New York, home, located near the T.J. Watson Research Center, where he worked for nearly four decades until his retirement.

Photo courtesy of IBM Archives

Beginnings

Computer: Let's start at the beginning. How and when did you initially become acquainted with computers? And when did they begin to fascinate you?

Cocke: Sullivan Campbell, who worked with IBM for several years, came to Duke after working on Oracle at Oak Ridge, Tennessee. Oracle was essentially a 40-bit-wide parallel von Neumann computer like the one at the Institute for Advanced Study. I had just graduated and had planned to spend the summer at Duke working for J.J. Gurgen, a mathematician who was hired to study which computers the Army might use. I rented a room from Sully Campbell—he was working for Gurgen too—and we drank a lot of beer and spent a lot of time talking about computers. I had studied mathematics but didn't know anything about computers until I sat around and talked to Campbell about Oracle, and building faster adders—simple-minded things like that. The field just interested me.

Computer: How did you come to work for IBM?

Cocke: I was thinking of going to work for Arthur D. Little, so I went by there and by GE. I had a friend at IBM—a logician named Brad Dunham—who took me around to see Steve Dunwell, the head of the Stretch project, which was just starting. Dunwell convinced me that working on Stretch would be interesting—one of the main goals was to make it run fast—and so I joined IBM in 1956.

We had a very interesting group that included Jim Pomerene—who built the CRT memory for the Institute machine, the Johnniac—and Fred Brooks. I had a desk in between them. I learned about Fortran from Irv Ziller, who worked on the original compiler and invented Fortran 3. Our team also had Gerry Blaauw and John Fairclough. John later worked as manager of the IBM Hursley lab and then for the prime minister, and he was later knighted.

I was delighted that the people I met knew something about computers because I didn't. Subsequently,

0018-9162/99/$10.00 © 1999 IEEE

I talked Sully Campbell into coming to IBM, which had an outrageous reputation in those days. At the time, IBM was basically a punch card business, so Campbell sent me up as a trial balloon. If I didn't quit immediately, he would take a chance on coming.

Influences

Computer: Seymour Cray frequently set the standard for competition in high-performance computing. Did understanding his machine design techniques give you any insights into what could or should be done in computer architecture?

Cocke: Yes. I always had the greatest admiration for Cray as a computer architect. He had a lot of good ideas, not just the 6600, but his earlier machines. He had progressive indexing and many other things that gave you high speed. I think he was a real computer man. He knew a lot about everything. I never met him or heard him speak, but some of those who did have told me he had a terrific sense of humor, which I didn't suspect.

Computer: Does anyone besides Cray fit your definition of a real computer man?

Cocke: Campbell around IBM knew a lot about various parts of computers, but not as much as Cray, who excelled at circuit design, logic design, packaging—everything. He built them, cooled them, and wrote his own operating system. I don't know anyone else like him. I wish I had met him; I'm sure Cray would have been great fun to talk to. I also liked that he wasn't afraid to start his own company.

Computer: Do you have any thoughts on how computer architecture should be taught today?

Cocke: When I started out, I worked with a lot of people who didn't take any courses in computer science, because there weren't any. Until Don Knuth came along, no one wrote any really good computer science books. After he wrote his fundamental-algorithms book, lots of other books came out.

I think you should—depending on how elementary the course—teach how computers work: logic, adders, ways to make them go faster, but especially memory. There's been a 10^7 improvement in cost-performance since Dennard invented DRAM. We used to get a dollar a bit for memory, and it's now down to about a dollar a megabyte. And performance is much better, too. We had 12-μs memory on the 704, and now we have 100-ns memory. And Dennard also worked on scaling—what happens when you shrink circuits in size and keep the power density constant. That work was a major contribution. So we have Dennard to thank for two major accomplishments.

Computer: You seem to admire Don Knuth. Did your many talks with Knuth on computer-related issues have an impact on your thinking?

Cocke: Absolutely. For example, before profiling, I thought we should compile in counts and understand the frequencies. He gave a course about profiling where he got frequencies and had schemes that allowed compiling in fewer counts, and calculated the rest. He also knew a lot about compilers and languages—how many doubly subscripted, singly subscripted, and unsubscripted variables there were in Fortran programs and so on. We agreed that developers should have access to a lot of facts when working with compilers. Knuth is a hard-working, organized man, and I feel he made more of a difference in the computer industry than anybody.

System design

Computer: You've designed several machines that have made a tremendous impact on current processor design, including Stretch; ACS; and the 801, RS/6000, and PowerPC processors. Let's talk about them.

Stretch

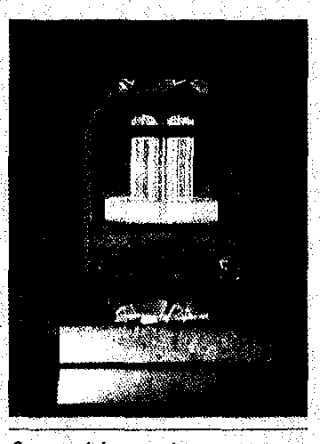
Stretch may have been the first machine to include partitioning of the instruction execution process into the instruction fetch-decode phase and the data execution phase. This development gave rise to instruction look-ahead, pipelining, short-circuit data forwarding,

The Seymour Cray Computer Science and Engineering Award

Seymour Cray, a contemporary of John Cocke, has long been regarded as the father of supercomputing. In 1997, to honor Cray's legacy, the IEEE Computer Society and Silicon Graphics Inc. established an annual award recognizing innovation in high-performance networking and computing. Cocke was chosen this year as the first recipient of the Seymour Cray Computer Science and Engineering Award.

Cocke's Cray Award citation reads, "for unique and creative contributions to the computer industry through innovative high-performance system designs." He will receive a crystal memento and an honorarium of $10,000 endowed by a gift from Silicon Graphics, the current producer of Cray Supercomputer products.

Seymour Cray



Cray crystal memento

partitioned memory, and instruction backout to preserve hard interrupts. What led you to develop these innovative features?

Cocke: Stretch was a joint project between Los Alamos and IBM. During our discussions with them, we gave a sense of how the timing on Stretch—and more particularly look-ahead—would work. I said I'd just write a simulator for it in Fortran. To write the simulator, I had to dream up what look-ahead would be like: When you gave a load, it was not held up by the next op; you gave a load, then you gave an add, and the load would load a buffer, the add would load a buffer, and the actual operation would execute. But the next op was not held up by the add being executed on the basis that it was tied to the reference from memory. So you put an op in a buffer to be executed later and went ahead and loaded the data into a buffer, too.

Computer: Stretch's goal was to be a hundred times faster than the 704. To achieve speed required a complex machine that, in turn, required a complex simulator. How practical was it to run code sequences through that kind of simulator so they would be of any value to you?

Cocke: The sequences made it clear that we were not going to make that performance in good time. The simulator helped us find out a lot of information about

Project Stretch

Werner Buchholz

Project Stretch developed the first commercial supercomputer—the IBM 7030 or, less formally, Stretch. Stephen W. Dunwell at IBM led the effort, which began in 1955 and proceeded in close cooperation with the Los Alamos Scientific Laboratory. The first system was delivered there in 1961, a year late, but it was up and running reliably a month after arrival and remained in use for the next 10 years.

Project Stretch was a highly ambitious attempt to "stretch" the available technology beyond what would have been the usual next step in development. At a time when all electronic computers in production were still using vacuum tubes, Stretch was to be an all-solid-state machine with high performance and reliability. The performance goal was 100 times that of the fastest machine IBM had in production—the 704.

In addition to the high-speed drift transistors, circuit boards, and magnetic core memory that were developed for the project, Stretch pioneered a number of techniques and concepts, including

- memory addressing in power-of-two increments (with 64-bit words for high-speed arithmetic and 8-bit bytes for input/output);
- 72-bit words in memory, including 8 bits for automatic error detection and correction;
- magnetic disks instead of drums for secondary storage;
- provisions for safely running multiple programs simultaneously; and
- a standard interface for attaching different I/O devices.

Stretch also introduced some new terminology—*byte* and *system architecture*—now in common use. John Cocke's two major contributions were the design and implementation of a detailed timing simulator and an instruction look-ahead, which permitted overlapped operation within a single instruction stream.

Nine systems were built, including one that was part of a special product for the National Security Agency, called Harvest.

Photo courtesy of IBM Archives

Because Stretch's actual performance did not meet expectations, IBM lowered its price to the point where the product would not be profitable and stopped marketing it. But Stretch did contribute much technology to what proved to be commercially successful IBM products, such as the 7090 computer. The later System/360 carried forward many of the concepts from Stretch, including that of a line of compatible machines, which was planned but never implemented for Stretch.

Further reading

C.J. Bashe et al., *IBM's Early Computers*, Chapter 11, MIT Press, Cambridge, Mass., 1986.

W. Buchholz, ed., *Planning a Computer System—Project Stretch*, McGraw-Hill, New York, 1962.

***Werner Buchholz** contributed to a number of IBM systems including the 701, 702, Stretch, and System/370. He retired from IBM in 1987.*

the machine's design and uncovered several bad ideas. One of the worst ideas we found involved having registers be addressable as part of memory. The justification for that was to make things "clean"—that's the worst word in computer architecture. Say we had a store instruction. You can't figure out what is affected by the store. This screwed up all hopes of writing a decent register allocator.

Computer: Given all of Stretch's unique features that are so important today, such as the partitioning of instruction execution, the execution process, short circuiting, look-ahead, partitioned memory, and so forth, which feature do you consider most important?

Cocke: That's hard to say. Partitioned memory was sort of a default. We had a very fast 1,000-word memory, and we decided to partition it to use its performance, as well as have a much larger memory to keep our instructions in.

Computer: Was partitioned memory a result of the instruction look-ahead that let you keep fetching instructions?

Cocke: We felt that partitioned memory would not destroy look-ahead performance, though it actually did slow a Monte Carlo simulation we had that was full of branches. So when we got to ACS, we incorporated some very fancy branch prediction. We had a prepare-to-branch instruction and added skips—skip on bit, skip on no bit—whereby you could mark an instruction with a bit, and conditionally execute it. You could prepare to branch, execute as many instructions as you wanted, come to this bit, and then go. If we hadn't prepared the branch, we couldn't have an instruction set up for skipping. Branch preparation let us take some of the burden off branches.

Advanced Computer System

Computer: ACS was perhaps the first project that pushed very hard simultaneously on all aspects of its design: the machine organization, the compiler, the packaging technology, and reliability. Did the experience with Stretch lead you to realize that ACS's compiler and hardware should be designed in tandem?

Cocke: When we first started the ACS project, one of our main concerns was when we would do the instruction set design. As we wrote the compiler, we wanted to make sure it provided consistently good compilation of the instructions we were providing.

We worked on things like the number of registers, particularly things dependent on the architecture of the machine, but also machine-independent things like common subexpression elimination. We worked on reduction of strength. That was a bad name; we should have called it "Babbage differencing." In other words, when you're calculating a subscript, as you are going around the loop as you increment *i*, for instance, all you have to do is add the dimension of *a* times *i* to get the subscript updated. You don't have to do *i* times multiply and calculate it all. That was independent of the machine to the extent that you assume that multiply was slower than add.

Computer: Given that you undertook development of the optimizing compiler and machine design simultaneously, did the results of the compiler impact the design of the machine and vice versa?

Cocke: Yes. We found that we could do a lot of things that we didn't think practical at first. Cray had progressive indexing—when you give a load, you increment the address pointer—and we had that too. We thought that it might be hard for the compiler to implement and found it wasn't. We would use the same algorithm for reduction of strength and so forth. We got a lot of ideas from the 704 compiler, the first Fortran compiler. It didn't have subroutines but did have a lot of optimization and was terrifically clever. In fact, some of the code it produced was so good that I was reading the object code and thought it had a bug, but then realized it was just amazingly clever.

When the queen asked who was second, the observer replied, "Madam, there is no second."

Computer: One of ACS's most important concepts is the decoding and issuing of multiple instructions per cycle. How did you arrive at this idea?

Cocke: I credit Gene Amdahl with that idea. He wrote a paper that said the fastest single-instruction-counter machine has an upper bound on its performance. I wanted to make a faster machine. So we looked at his paper, which said you can only decode and issue one instruction per cycle, and we decided to get around that limitation. The paper helped us a lot because, even though many of his hypotheses were wrong, it helped us see them. He appreciated that all computers at that time had certain properties that prevented them from going faster. So we designed a machine with different properties.

RISC: 801, RS/6000, PowerPC

Computer: You also worked on the 801. I believe it was code-named America after the first America's Cup Race off the coast of Wales, and is widely regarded as the first RISC processor.

Cocke: Yes, I actually gave it that name, based on a story I heard. While attending the first America's Cup Race, Queen Victoria had some gentleman observe the race's progress for her. As the lead ship rounded the island, Victoria asked, "Who is first?" "America," her observer replied. When the queen asked who was second, the observer replied, "Madam, there is no second." That's why I picked the name.

Computer: What was the impetus behind the project?

IBM's ACS-1 Machine

Ed Sussenguth

IBM's System/360 line of computers was announced in April 1964. It was expected to do well in commercial markets, but its attractiveness to the scientific market was uncertain. Several projects were started to address the high-performance scientific market. Project X began in 1961 and became the 360/91. Project Y was begun as a non-System/360 compatible research effort in late 1963, and in the summer of 1965 it was transferred to a development organization and renamed Advanced Computing System 1 or ACS-1.

Project Y's goal was 1,000 times the performance of the IBM 7090 or about 160 million instructions per second. That goal was to be achieved in several ways. The 7090's hardware ran at 20 nanoseconds per circuit; ACS's would run at 1.6 nanoseconds—a factor of 12.5. An additional factor of four was required from the machine architecture and organization. To achieve this, many independent units were to operate concurrently: instruction fetch, instruction decode, data fetch and store, index arithmetic, floating-point arithmetic, and branch calculation. Moreover, multiple instructions were to be executed on each machine cycle. The remaining overall improvement was to be achieved by additional circuits providing parallelism and optimizations to be done in the compiler.

ACS had numerous innovations; here I list a few of the highlights. Key among these was the instruction sequencing logic, which examined eight instructions per cycle and dispatched as many as seven for execution, possibly out-of-order. Branch instructions were overlapped with computation, with prefetching down multiple paths, and a deferred-branch architecture. Backup registers provided a precursor to various register-renaming schemes.

Among John Cocke's key contributions to the ACS effort was his insistence that the compiler, architecture, and machine organization be developed in concert. To aid this, the performance of ACS was simulated during development on a cycle-by-cycle basis. This allowed a complete source program-to-timing analysis view of the operation of real problems with the design.

Delays in the development of the ACS circuit family, packaging problems, and an excess number of circuits forced ACS-1 to continually slip its original schedule. Moreover, it was determined that most of the ACS-1 innovations were applicable to 360-class computers, and the line of 360 computers was selling well. In 1968, ACS-1 was phased out.

Further reading

H. Schorr, "Design Principles for a High-Performance System," *Symp. Computers and Automata*, New York, Apr. 1971, pp. 165-192.

M. Smotherman, "IBM Advanced Computer System—A Secret, Scientific, Superscaler Supercomputer from the 1960s," http://www.cs.clemson.edu/~mark/acs.html.

Ed Sussenguth was a major architect of the ACS project and later contributed to IBM's Systems Network Architecture.

Cocke: At the time, IBM and LM Ericsson were discussing a joint project to develop a controller for a telephone exchange. We were going to build a time-division switch and control it with the 801. In the end, though, the project fell through.

Computer: Was the 801 effort prompted or influenced by experiments that indicated the System/360 compilers only generated a subset of the System/360 instruction set?

Cocke: Yes. One thing we did with RS/6000 was try to make sure the compiler could easily generate every instruction we had. We took the point of view, which you should always take, of being flexible. Initially, we said every instruction was simple: It did something, not something and something more. Then we said, the heck with that, why not have it do *and*—branch and count *and* set a bit. You start off and clear a bit. First time through a loop, you did the count and set a bit so that you weren't delayed by the time it took to do the count, and the next time around the loop, you test the bit and take the branch.

Computer: What was the major architectural difference between the 801, the RS/6000, and the PowerPC processors that evolved from the 801? What motivated those changes?

Cocke: Well, IBM Austin started to build a thing they called ROMP, which came out of the 801 effort. We went down to Austin, and they asked us to build a floating-point machine. We threw out ROMP and designed a floating-point machine that became the RS/6000.

Computer: In addition to the floating-point unit, what were the RS/6000's other architectural differences? Could you do different multiple issues, different decodings?

Cocke: Well, the opcodes were vastly different. We had a triple address machine and 32 registers. We'd also learned from Stretch to avoid indexing through registers.

Computer: Stretch and the ACS can be described as having a good deal of complexity in the hardware; they also implemented rather complex instructions. The 801 and RS/6000 processors contained much less hardware complexity, which implemented much simpler instructions and relied increasingly on more compiler knowledge. So we can see an evolution from complex machines and semi-intelligent compilers to simple instructions and very intelligent compilers. Can you tell us how your thinking evolved over those three or four machines and compilers?

Cocke: We saw that we could create—and had, by

the time RS/6000 came along—a very good compiler. Nowadays, everybody in graduate school takes a course on how to write compilers. Now, courses cover every kind of optimization, including a lot of things we didn't do. We didn't do profiling, which in retrospect we should have done. Profiling involves the operating system—you have to go beyond the compiler. You want to compile, run, and count frequencies, and then decide where to emphasize your optimization.

Computer: Are you suggesting that the compiler design really involves the concurrent evolution of the compiler to both the hardware and the operating system?

Cocke: If you want to do things like profiling, you want the operating system's cooperation. Now, a lot of people want interpretation as part of that cooperation. You interpret and count, then compile. People have all kinds of opinions on this, and you can choose from a great number of approaches. You can scan the program every time, keep going and execute, or you can scan and translate it into an intermediate language, then interpret that intermediate language. Once you have done the interpretation, you can compile.

Computer: Given this range of approaches, which do you support?

Cocke: I favor translating into an intermediate language, interpreting, getting a count, and then compiling it.

Parallel computing

Computer: Some in the industry have ascribed to you the ambition of designing the fastest possible uniprocessor. Some have even suggested that you may be ambivalent about designing and implementing parallel computers. What are your views on designing parallel systems?

Cocke: When you couple machines tightly, you get a lot of interference, and the addition of the second machine doesn't come close to doubling performance. With closely rather than tightly coupled machines, it's easier for the hardware. The SP2s do a decent job in achieving a fair percentage of two machines' combined performance. Ramesh Agrawal has programmed a multidimensional Fourier transform that performs better as you get up to higher and higher dimensions. He's also done this excellent bucket sort that gives you almost linear performance on multiple machines. But all those enhancements don't come easily.

Computer: What conclusions do you draw from this situation—that it's just algorithm-smart programming?

Cocke: If we build the right type of communication between the memories of the various processors, we will ultimately obtain good performance. But in the end, it's the algorithms: If you look at things like fast Fourier transform, it went from N squared to $N \log n$. There is no way you can make an improvement in computer architecture that comes close to that.

Computer: Do you think parallelism, say, in pairs of multipliers is useful?

Cocke: Well, the utility of the second multiplier isn't as valuable as the first, and by a large amount. I don't remember the figures, but they aren't good.

Future architectures

Computer: So, how would you summarize this situation? Can we expect technology or architecture improvements that achieve 10 to 20 percent or—if we're very lucky—50 percent per generation?

Cocke: If, for instance, you make an improvement in the compiler and you get a 10 to 15 percent performance boost, you would consider that a very good result. You might get a multiple of that performance boost in hardware by making your CPU clock run a lot faster. I believe we are going to be doing that in the next few years.

Computer: Where do you see the clock speeds going in the next few years?

Cocke: Up to 100 gigahertz.

Computer: We are currently close to one gigahertz, and so you are projecting a two-orders-of-magnitude improvement?

Cocke: In 15 years we will get there. It's still just a multiple, not large. The hard part will be a 100-picosecond clock distribution without skew, but we have some good ideas about how to distribute the clock.

IBM has copper wiring. Several years back, I saw a photomicrograph of wiring that looked like the framework of a skyscraper, with the wires in the air. We'll have X-ray lithography so we can get normal scaling rules, or better. As you shrink the chip, the scaling and cooling get better, so we can let power density grow by cooling the chip. Thus, we will have less capacitance and less resistance—that is, very low RC time constants—so that performance can go way up. We will also have good programs for optimizing the circuits; it's quite clear that as circuits become faster, it's easier to make them faster still.

Say I use X-ray lithography to make a chip 3mm^2, so that I have a four-processor CPU with a 3-mm^2 cache. I'll need to worry about main memory scaling in performance, but if I map this 3-mm^2 chip on a memory chip that's, say, one centimeter or two centimeters on a side, that gets me to 256 Gbytes. So I have a lot of memory, and I can make fast memory accesses—say 5-ns memory. That's 50 cycles. That's straining cache a little, but it's close to being okay. Then I read out interleaved and so forth and have what they call nonblocking cache—that is, a queue

Clock speeds will climb to 100 gigahertz in the next 15 years.

801 and PowerPC

George Radin

The 801 was a research project pursued in IBM between 1975 and 1980. John Cocke was the intellectual leader of the team of about 30 computer scientists and engineers that worked on the project. The project developed a processor architecture and design, a PL/I-like programming language and optimizing compiler, and an operating system. The overall goal was to exploit the synergistic advantages of designing all the components together in order to stretch the limits of uniprocessor performance within minicomputer cost constraints.

More specifically, the emphasis was on minimizing the processor cycle time, the average number of cycles per instruction, and the number of instructions executed. The system organization featured a simple, regular data flow to minimize the cycle time and to provide a good compiler target, and included separate caches for instructions and data. The instruction set incorporated delayed branching, multiple condition registers, instructions to allow the operating system to reduce unnecessary cache misses, and an interrupt definition that avoided the need to flush the pipeline across interrupts or state changes.

In addition to pioneering a number of program-rewriting and code-motion optimizations, the compiler used a coloring algorithm for register allocation. The compiler generated very simple code from the source program—that is, code for an architecture similar to the target architecture, but with an infinite number of registers. This intermediate code was then optimized. The simplicity of the code and the manner in which address calculation was exposed made the optimizations very effective. Register allocation then produced the ultimate, executable code.

The 801 research effort led directly to the Power architecture used in the RS/6000. That in turn formed the basis for the PowerPC architecture now used in the RS/6000 and in Apple systems. A number of the innovations pioneered in the 801, some of which first appeared in John Cocke's earlier systems, have influenced many other efforts. Versions of the 801 were used in mainframe I/O processors and in the IBM 9370. It also seems clear that, by the time the 801 project was completed, several 801-based ideas were being independently investigated by other groups. The acronym *RISC* was coined by researchers at UC Berkeley to describe processors built on similar principles.

Further reading

J. Cocke and V. Markstein, "The Evolution of RISC Technology at IBM," *IBM J. Research and Development*, Jan. 1990.

G. Radin, "The 801 Minicomputer," *IBM J. Research and Development*, May 1983, pp. 237-246.

George Radin was the manager of the 801 project. He also contributed to the PL/I language.

for my cache—so that makes it a little bit better. I'm not saying that IBM will achieve this immediately, but five or 10 years is a very long time in computing.

Reconfigurable computing

Computer. What does the future hold for reconfigurable computing systems?

Cocke. I'm a great believer in them, because that's the way to build a simulator fairly fast. But if I consider the domain and range, let's say I have a 32-bit word, how many functions do I have? The answer is approximately 2^{70}, which is a lot. Take a normal 32-bit adder, what functions do I have? Add, subtract, multiply, divide. That's what math is. So how do I think of 2^{70} useful binary combinations? Reconfigurability won't be used in the functional unit, to do a square root, but it will be good for a hashing scheme or encryption—70 bits is a long-enough key.

Reconfigurability is also good for control. Say I want to run a dataflow machine or something. I have a machine like an evaporator or etcher that is automatically controlled. If I give you the instructions, how long does it take you to build a computer? But if I have this reconfigurable machine, and I debug and assemble it and tell this other machine to build it. Say it takes two weeks to make the machine. So what? Big problems like that take a lot longer.

Quantum computing

Computer: What about computing at the molecular level?

Cocke: I believe the situation's like this: In quantum computers, P is equal to NP. Now, that's only a theoretical result, and nobody knows how to build these quantum computers, but they're working like hell to figure out how to do it. I saw an article in *Physics Today* on how you can use a laser to shape the wave function of an electron. Somebody will obviously figure it out—remember that Kelly at Bell Labs said build a transistor, and Shockley, Bardeen, and others did it. I don't know how it will work out, but people are a lot more sophisticated today, and so eventually we will have quantum computers.

If P is equal to NP, that is a giant difference. If I have a proof checker, for example, I scan over the proof and I check it and make sure it's okay. Now, the dif-

ference between a proof checker and a theorem prover is the difference between P and NP, right? I mean, that is fantastic. It's like you start out trying to prove everything in every direction, and whenever I find the one that's been checked, I say here is a proof. But with a molecular computer I'm proving everything, right? That's a Turing machine.

Just curious

Computer: You have been described as having a very fertile mind, inspiring and guiding your colleagues both within and outside of IBM. What would you advise others to do as mentors and catalysts in industries and universities?

Cocke: I just do what I do because I am interested in computers. I don't have a plan to make myself effective. Almost nothing about computers escapes my attention, including the people interested in them. I can't say that I have any systematic idea of how to move things along.

Computer: Would you describe yourself as driven?

Cocke: No. I just get along with a normal amount of curiosity.

Computer: Is that your main motivator? Curiosity?

Cocke: Well, I don't know. My father, when he graduated from law school at something like 18, went to work for his uncle's law firm. He couldn't take the bar exam because he wasn't old enough. So he was kind of like an office boy—and they teased him a lot about his curiosity. One day, he went by a drugstore, saw a black box with a hole in it, and wondered what it was for. He stuck his finger inside and got a bad cut. Turns out the box cut off cigar ends—something my father found out the hard way. I think I've inherited my curiosity from him. Hopefully, I haven't had quite as bad an experience with it, though.

Computer: Is there any advice you would like to give the next generation's computer architects?

Cocke: I don't know. I do what I do, and I don't plan how I ought to do it. I never have. I don't believe in being rigid about anything. If I see an opportunity, I will drop all the rules, even when doing so is probably a mistake. ❖

Acknowledgments

We thank Vicky Markstein of SilverMark, Chris Sciacca of Technology Solutions, Takako Yamakura of IBM, and Robert Godfrey of the IBM Archives for their help in locating images.

Bruce Shriver consults for Genesis 2 Inc. He is a professor-at-large at the University of Tromso in Norway and an honorary professor at the University of Hong Kong. Shriver is a Fellow of the IEEE and a past-president of the Computer Society. Contact him at b.shriver@computer.org.

Peter Capek is a research staff member at the IBM T.J. Watson Research Center and a member of the IEEE. Contact him at capek@us.ibm.com.