The IBM 650: An Appreciation from the Field

DONALD E. KNUTH

Editor's Note

How does one summarize the personal reminiscences of another of the giants in the computer field? Don Knuth is also an artist, of course, as witness his comments on Poley's code and his "Art of Computer Programming." Knuth was a tremendous help to me in preparing this special issue. He wrote to other participants and encouraged them and me. Here is a letter he wrote me as this project was starting:

Dear Cuthbert,

When you asked if I might be interested in writing something about the IBM 650, I thought I might be able to come up with about two pages worth of stuff. But when I began to reminisce, it became clear that I should write about ten times as much as I had originally thought.

Here is the result; I hope you like it. Tears ran from my eyes as I (sob) wrote the conclusion!

I suppose it was natural for a person like me to fall in love with his first computer. But there was something special about the IBM 650, something that has provided the inspiration for much of my life's work. Somehow this machine was powerful in spite of its severe limitations. Somehow it was friendly in spite of its primitive man-machine interface.

I had just turned 19 when I was offered a part-time job helping the statisticians at Case Institute of Technology. My first task was to draw graphs; but soon I was given some keypunching duties, and I was taught how to use the wondrous card sorter. Meanwhile a strange new machine had been installed across the hall—it was what our student newspaper called "an IBM 650 Univac," or a "giant brain." I was fascinated to look through the window and see the lights flashing on its console.

One afternoon George Haynam explained some of the machine's internal code to a bunch of us freshmen who happened to be in the lab. It all sounded mysterious to me, but it seemed to make a bit of sense, so I got hold of a couple of manuals. My first chance to try the machine came a few weeks later, when one of the upperclassmen at the fraternity I was pledging needed to know the five roots of a particular fifth-degree equation. I decided that it would be fun to compute the roots by using the 650. More precisely, I had been reading the manual for the Wolontis–Bell Labs Interpreter [see Technical Newsletter No. 11 in this issue], and I decided that polynomial root finding would be a good test case.

A program for the Bell System (as we called it) consisted of 10-digit numbers like

1 271 314 577

which meant "Add the (floating-point) number in location 271 to the (floating-point) number in location

Note: The preparation of this paper was supported in part by National Science Foundation grant MCS-83-00984.

Author's Address: Computer Science Department, Stanford University, Stanford, CA 94305.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**], Single Data Stream Architectures; K.2 [**History of Computing**]—*hardware, IBM 650, people, RUNCIBLE, SOAP, software. General Terms:* Design, Languages. *Additional Key Words and Phrases:* education.

314 and put the result in location 577." I found a book that gave formulas for the roots of a general fourth-degree equation; so it was easy to factor a general real polynomial of degree 5 by first doing a simpleminded search for a real root r, then dividing by $x - r$ and plugging the result into the formulas for quartics.

I realize now how lucky I was to have had such a good first encounter with computers. The polynomial problem was well matched to my mathematical knowledge and interests, and I had a chance for hands-on experience, pushing buttons on the machine and seeing it punch the cards containing the answers. Furthermore, the Bell language was an easy way to learn the notion of a program that a machine could carry out. I've forgotten the name of the fraternity brother who asked me to solve this particular problem, but I bet he's kicking himself now for not having done it himself.

I often wonder whether it might not still be best to teach programming to novices by starting with a numeric language like that of the Bell interpreter, instead of an algebraic language like BASIC or LOGO. I think a small child can understand machinelike language better than an algebraic language. But I know that such ideas are now considered out of date, and I suppose I'm being an old fogy.

I learned a few years ago that the Bell interpreter had been inspired by John Backus's Speedcoding system for the IBM 701 (Backus 1954). During my student days I had never heard of the 701, and this, I think, leads to an important point: The IBM 650 was the first computer to be manufactured in really large quantities. Therefore the number of people in the world who knew about programming increased by an order of magnitude. Most of the world's programmers at that particular time knew only about the 650, and were unaware of the already extensive history of computer developments in other countries and on other machines. We can still see this phenomenon occurring today, as the number of programmers continues to grow rapidly.

When I did finally learn about the existence of the IBM 701, it had been improved to the 709, and it was shortly to become the 7090; but I must confess that I still liked my good old 650 a lot better. The 650 had only 44 operation codes (IBM 1955) [see "Optimum Programming" in this issue], while the 709 had more than 200; yet I never enjoyed coding for the 709, because I never seemed to be able to write short and elegant programs for it—the opcodes didn't blend together especially well. By contrast, it was somehow quite easy and pleasant to do complex things on the 650 with very few instructions. Most of the commands in the 650's repertoire accomplished several things at once, and it was frequently possible to make good use of the side effects. For example, the instruction

```
60  1234  1009
```

meant, "Load the contents of location 1234 into the distributor; put it also into the upper accumulator; set the lower accumulator to zero; and then go to location 1009 for the next instruction." All four of these actions were often useful in the subsequent program steps.

In fact, I usually got by with only 34 of the 44 opcodes, because I seldom had a good application for the ten "branch on distributor digit equal to 8" commands. After 25 years I still can remember the numeric codes for most of the remaining 34 ops; and I'll never forget the fact that addresses 8001, 8002, and 8003 referred to the distributor, lower, and upper accumulator registers.

The 650's "one-plus-one address" code, in which each instruction designated the location of its successor (and branch instructions designated both successors), has been rejected by modern machine designers. But it was in fact extremely effective, because it allowed convenient subroutine linkage and because it became easy to execute instructions from registers. A one-plus-one scheme was important, of course, on a machine without efficient access to all words of memory, because instructions could be located in "optimum" places on the magnetic drum.

The incredible thing about the 650 was that we could do so many things with it, although it was three orders of magnitude slower than today's computers, and it had three orders of magnitude less memory. The memory-space limitation was more important than anything else during my first year of programming. I had to learn how to pack data and how to use subroutines in order to save space. For example, my first large program was a tic-tac-toe routine that "learned" to play by remembering the relative desirability or undesirability of each position that it had ever encountered. The hardest part was figuring out how to keep one digit of memory for each possible configuration of the board; board positions that were equivalent under rotation or reflection were considered to be identical.

The first program that I ever wrote in machine language still stands out in my mind. It was June 1957, and my freshman year at Case had just ended. I decided to hang around Cleveland instead of going home, and I was allowed to stay up all night playing with the computer by myself. So I decided to write a program that would find prime factors. The idea was that a person could set up a 10-digit number in the console switches and start my routine, which would

punch the corresponding prime factors on a card and stop; then another number could be set up and factored in the same way, etc. I believe my first draft program was about 80 instructions long, but I didn't save it, so I can't be sure. Anyway, I wrote it as a sequence of about 80 decimal numbers, and punched it onto cards—much as I had done with my previous (Bell system) program for root-finding. Then I sat down at the console of the machine and began to learn how to debug, using the half-cycle switch to step through the instructions slowly, or using the address-stop switch to discover when the program used particular locations for data or instructions. The 650 console was excellent for on-line debugging, and nobody else was using the machine at that time of night.

Well, my program was riddled with errors, and I removed them one by one during the next two weeks. Besides the "obvious" mistakes, I hadn't realized at first that a 10-digit number can have as many as 33 prime factors. Only eight numbers could be punched on a card, so I would have to punch up to five cards. (My original program had only thought of punching one card.) Then I had to clear the memory between runs so that spurious data from a previous factorization wouldn't appear on the next one, and so on. You know the story; we all make the same mistakes. I was lucky enough to have the opportunity to make lots of mistakes right from the beginning, and to diagnose them all by myself, sitting at the machine. All the facts I needed were available to me, because I was working in machine language, and no operating system or other software was interposing itself betwen me and what I needed to know. Debugging took a long time at first, but I think I had the machine to myself about six hours every night. Finally I arrived at a program that was satisfactory; I vaguely recall that it took about 11 minutes to determine that the number 9999999967 was prime, although at one point this particular test case had taken 17 minutes.

By this time my program had grown to 140 words long, and I think I had changed each of the instructions at least twice. I had also learned about the SOAP assembly language (Poley and Mitchell 1955), so my final program was expressed in symbolic form; I had been weaned away from numeric machine language during those two weeks. The success of this program gave me the confidence to try another (which converted a given number on the console switches to a specified radix); then I was ready for tic-tac-toe.

The SOAP language allowed symbols to be up to five letters long, and I recall spending a lot of time trying to come up with suitable names. It was a great moment when I hit on the right term for the program step that was to be executed when the computer had won at tic-tac-toe by finding three x's in a row: I called that step BINGO.

I regret to report that I've recently looked again at my prime factors and tic-tac-toe programs, and they are entirely free of any comments or documentation.

Shortly afterward I got hold of the SOAP II manual (Poley 1957), which impressed me greatly and had an enormous influence on my subsequent career. This manual included the entire listing of SOAP II in its own language, and the program was absolutely beautiful. Reading Poley's code was like listening to a symphony; I wanted to be able to compose programs like that. I also learned several new techniques, such as hashing, from this code. My next project was to write a modification of SOAP II that would have worked on a 650 with only 1000 words of memory. (I knew that such machines were sold, but I never actually saw one.) Then I spent the rest of the summer writing SOAP III (Knuth 1958), which went the other way by adding additional features for enhanced 650s that had index registers and/or floating-point hardware.

SOAP III was my introduction to software writing. In particular, I learned about what is now called "creeping featurism," where each of my friends would suggest different things they wanted in an assembler. I probably tried to accommodate them all, since SOAP III had 24 pseudo-operations that were not in SOAP II. I also left 150 memory locations available for user-defined pseudo-operations. I put liberal comments into the code, having learned that lesson at last.

Our lab received an amazing program from Carnegie Tech during the summer of 1957, namely the famous IT compiler by Perlis and Smith (1957). IT took algebraic statements as input, then computed awhile, and punched SOAP programs as output. I had no idea how such a feat would be possible, but I got hold of the program listing at the end of the summer, and I read it while vacationing with my parents at a beach resort on Lake Erie. This program was not beautifully written like Poley's, but it accomplished remarkable things, so I naturally had an urge to rewrite everything in the style of 650 coding that I had just learned. Bill Lynch and I began this project late in 1957, under the direction of Fred Way III and George Haynam. We first called our program Compiler III, but it eventually became known as RUNCIBLE (Knuth 1959*b*; Case 1959). The language was a superset of Perlis's IT, and we worked very hard to squeeze in as many new features as we could.

Somehow it was possible to cram a complex compiler into the 2000 words of the 650. Yet when we were done, I don't think we could have gotten by with only 1999 words, because we had spent considerable time finding every last bit of space—by using terrible

tricks so that small changes to one part of our code would usually cause some apparently unrelated part to blow up. I guess Parkinson's Law applies to programs as well as to organizations; we kept adding features until the space was filled.

RUNCIBLE had four versions called AX, AY, BX, and BY, where X stood for object code that invoked subroutines for floating-point arithmetic, while Y stood for object code that used the 650's optional floating-point hardware; A stood for SOAP output, while B stood for directly loadable machine-language programs punched five per card (and bypassing the need for assembly). It turned out that the X version became a Y version by replacing exactly 95 instructions by 95 others; similarly, the A version became a B version by replacing exactly 406 instructions by 406 others. If we discovered a way to save one line of code in, say, the A version, we looked closely at the B version until we had saved a line there, too.

We called the A version "two-pass operation," while the B version was called "one-pass." At the end of the summer I hacked together a "zero-pass" version that took one less pass than B, since it loaded machine instructions directly into their memory locations instead of punching anything on cards. For this I had to eliminate the matrix feature of IT; that is, doubly subscripted arrays were not permitted in "RUNCIBLE zero." My main goal was to prove that 2000 words of memory were not too few for a compile-load-and-go system, because somebody (Perlis?) had reportedly said that it would be impossible.

By 1959 our lab had acquired the ultimate in 650 upgrades: we had a full 653 system (IBM 1959) including index registers, floating-point hardware, and 60 whole new words of core memory! It was heavenly. Besides this, we put our printer on-line (so that listings didn't have to be made via cards), and we acquired a RAMAC disk storage, as well as several tape units.

At this point it was desirable to have a new assembly program so that we would make proper use of the new equipment. I therefore wrote SuperSoap (Knuth 1959*a*), a major improvement over SOAP III. I'm still pretty proud of SuperSoap, because it introduced some good ways of dealing with programs that would be loaded into the drum but executed from core, and because I had the courage to remove some features of SOAP III that didn't work as well as planned. Furthermore, SuperSoap introduced what I think was the best approach to the problem of "optimizing" the drum locations of data and instructions for the 650; it was a combination of machine and hand methods (Knuth 1961).

The name SOAP stood for Symbolic Optimal Assembly Program, and *optimal* meant that the machine would choose drum locations so that at least one reference to that location would involve no delay. Such optimization was much better than random placement of instructions; I had (for fun) experimented with SHOAP, a "Symbolic Horribly Optimizing Assembly Program" that used the algorithm of SOAP in reverse so that at least one reference to each location would lead to a 49-word-time delay. By adding seven cards to the normal SOAP program deck, you had SHOAP, which produced extremely slow programs. Conversely, it was possible to improve significantly on SOAP's performance by choosing locations carefully by hand and rewriting the program when necessary, as I discuss in Knuth (1961); the Bell interpretive system had been hand-optimized in a particularly beautiful way, which was quite an inspiration to me. In 1958, I wrote HAND SOAP, which permitted me to hand-optimize the locations without giving up the advantage of symbolic assembly. We used HAND SOAP to prepare the runtime system for RUNCIBLE; SuperSoap was later designed to incorporate similar ideas into a full-fledged assembler.

Somebody in 1958 or so circulated a joke about a program called RINSO, a "Real Ingenious New Symbolic Optimizer"; we were carried away by acronyms in those days. For some reason there has been an intimate relation between cleaning agents and software that I have written through the years, even though my programs haven't always been very clean. For example, John McNeley and I devised a system called SOL in 1963 (Knuth and McNeley 1964), and when I visited Norway a few years later I learned that SOL is the name of a Norwegian laundry detergent. Even more amazing was that my MIXAL assembler language (Knuth 1968) turned out to have the same name as a popular detergent in Yugoslavia—although I had had no idea that MIXAL would even be a word in any language! More recently, I have learned that TEX is a brand name for toilet paper in Greece . . . but I am digressing.

My preface to the SuperSoap manual (Knuth 1959*a*) gives a glimpse into the mood that prevailed at IBM 650 sites during the late 1950s:

> Soap 3 was written attempting to get as many features into 2000 memory locations as possible, but SuperSoap was written under a different philosophy; speed was the prime consideration, and storage space was conserved only when speed was not appreciably decreased. A factor of roughly 3:1 in running time over Soap 3 has thus been obtained. . . . Some of the pseudo-op rules have become more logical thanks to Carnegie Tech's TASS [a competing assembler, written by Art Evans]. . . . Once again much gratitude must be given to the Case Computing Center for letting me chew up thousands of cards.

On rereading SuperSoap, I find most of it reasonably similar to today's assemblers except in one significant respect: We assumed in 1959 that the computer lab would be an "open-shop" operation in which any student could come in and take personal charge of the machines while running a program. Therefore the error messages in SuperSoap consisted of machine halts, and my manual gave the following advice for error recovery:

> SuperSoap believes that the best place to catch errors is during assembly, and so it will stop if it finds something amiss. . . . The offending card is the fourth-last card out if you clear the read feed. . . . To restart, correct the bad card, . . . reinsert it in the deck, and hit Program Start.

There was a keypunch right next to the console, so this was probably the most efficient way to get the job done in those days.

Cards, cards, cards; we used tens of thousands each day. The run-time system of RUNCIBLE had a debugging feature whereby you could turn the console knobs and get a card punched for every statement of your program that was being executed; or you could even trace every machine-language operation, with one card per instruction. The 533 Card Read Punch could produce 100 cards per minute, and it often did.

One of the nice things about the 650 and its peripherals was their robustness. Our computing center staff could safely let random students work all of the IBM machines, changing plugboard control panels, clearing the punch hopper, mounting tapes, fixing card jams, etc., without worrying that the machines would be ruined. (This was emphatically *not* the case for the Univac equipment in another part of our laboratory; those machines had been designed with the assumption of a trained operator in attendance, and I tended to break them accidentally every time I went nearby. If all computers had been like those, a lot of people like me would never have gotten a good start on the use of computers, because we would never have been allowed to touch them.) During all my experience with the 650 I can remember only two instances where the design could perhaps have been slightly more foolproof: Once I discovered a special case of the divide operation that put our machine into an infinite loop, restartable only by hitting Power Off. (Later I visited Carnegie Tech and tried it on *their* 650; it blew the fuse! Ah yes, those were the joys of student life.) The other time was when one of the tiny console display lights was broken; the glass was gone and two little wires were sticking out. I changed the display so that this particular light was off, then tried to pull out the broken bulb by grabbing onto what looked like a dead filament. This gave me quite a jolt, and I was sick for a day or so. Perhaps the machine was trying to fill my brain with advice about how to write better software; or perhaps it was trying to kill me.

By 1959 I had developed a pretty good style of 650 coding, and I must confess also being addicted to tricks. One of the competitions among students was to do as much as possible with programs that would fit on a single card—which had room for only eight instructions. One of the unsolved problems was to take the 10-digit number on the console and to reverse its digits from left to right, then display the answer and stop; nobody could figure out how to do this on a single card. But one day I proudly marched up to the machine and made a demonstration: I read in a card, then dialed the number 0123456789 on the console, and started the machine. Sure enough, it stopped, displaying the number 9876543210. Everybody applauded. I didn't explain until later that my card would display the number 9876543210 regardless of what number appeared on the console switches.

There's more to the story. Our machine had an extra set of console switches, which were called register 8004. (As far as I know, Case's 650 was unique with this particular feature.) It turned out that nine instructions on an extended 650 were sufficient to reverse the digits of a number, and the ninth instruction could be put into one of the sets of console switches. Therefore I was able to solve the problem without cheating (see the appendix following).

The dirtiest trick I ever discovered for the extended 650 was to use the instruction "shift and count by 9004" in a certain context. This one instruction caused four things to happen simultaneously: (1) the upper accumulator was shifted left by four digits; (2) the lower accumulator was set equal to 10; (3) the core memory "timing ring" was set to 9004; and (4) the overflow indicator was turned on. I had an application in which all four of those things were useful.

SuperSoap was the last "system" software I wrote for the 650, although I wrote many application programs during the following year. Then I graduated, and began to tackle other machines. My favorite computer for the next five years became the Burroughs 220, which was another joy to use.

A number of my classmates and co-workers at Case later became leading figures in other computing centers; they include Bill Lynch, Mel Conway, Joe Speroni, Gilbert Steil, Jack Alanen, Mike Harrison, and many others. Our incubation period with the 650 was the foundation of our later work. And the same is true for thousands of other people (such as Bob Floyd) who became intimately familiar with 650s at other computer centers.

What was it about the 650 that made our experiences such a good foundation for our later careers?

Surely I wouldn't recommend that today's software be produced as we did the job then; we would never advance very far past the rudimentary levels achieved in those days, if we remained rooted in that methodology. But growing up with the 650 gave us valuable intuitions about what is easy for a machine to do and what is hard. It was a great machine on which to learn about machines. We had a machine organization that was rudimentary but pleasant to use; and we had program masterpieces like the Bell interpreter and Poley's assembler, as examples of excellent style.

We were forced to think and to develop our abilities to make mental abstractions about control structures; these experiences seem to have made us better able to do complex things later, when the task became easier. I'm reminded that Edsger Dijkstra began his programming experience in a similar way (but on a different computer); he and Zonneveld wrote the first ALGOL 60 compiler in a strictly numeric machine language.

This article about the 650 has turned out to be largely autobiographical. The fact is, it's impossible for me to write about that wonderful machine without writing about myself. We were very close. (One night I missed a date with my wife-to-be, because I was so engrossed in debugging that I had forgotten all about the time. I'll never live that down.) The 650 provided me with solid instruction in the art of computer programming. It was directly related to the topics of the first two technical articles that I ever submitted for publication (Knuth 1959*b*; 1961). Therefore it's not at all surprising that I decided in 1967 to dedicate my books on computer programming

> " . . . to the Type 650 computer once installed at
> Case Institute of Technology,
> in remembrance of many pleasant evenings."

REFERENCES

Backus, J. W. 1954. The IBM Speedcoding system. *J. ACM 1*, pp. 4–6.

Case Computer Center Staff. March 1959. "Runcible I." Case Institute of Technology Computer Center Reports, Series 5, Vol. 1, 67 pp.

IBM. June 1955. "IBM 650 Data-Processing System, Manual of Operation." IBM Corporation, Form 22-6060-1, 111 pp. (This was described as a "major revision" of forms 22-6060-0 and 22-6149-0, copyright 1953 and 1955.)

IBM. June 1959. "Immediate Access Storage, Indexing Registers, Automatic Floating-Decimal Arithmetic, and Magnetic Tape." IBM 650 Data Processing Bulletin, IBM Corporation, Form G24-5003-0, 48 pp.

Knuth, Donald E. February 1958. "Case SOAP III." Case Institute of Technology Computer Center Reports, Series 4, Vol. 1, 28 pp.

Knuth, Donald E. August 1959. "SuperSoap." Case Institute of Technology Computer Center Reports, Series 4, Vol. 2, 55 pp.

Knuth, Donald E. November 1959. RUNCIBLE—Algebraic translation on a limited computer. *Comm. ACM 2*, 11, pp. 18–21.

Knuth, Donald E. 1961. Minimizing drum latency time. *J. ACM 8*, pp. 119–150.

Knuth, Donald E. 1968. *The Art of Computer Programming.* Vol. 1, *Fundamental Algorithms,* Reading, Mass., Addison-Wesley, xxii + 634 pp.

Knuth, Donald E., and J. L. McNeley. 1964. SOL—A symbolic language for general-purpose systems simulation. *IEEE Trans. Electronic Computers EC-13*, pp. 401–414.

Perlis, A. J., and J. W. Smith. April 1957. "A Mathematical Language Compiler." In *Automatic Coding,* Philadelphia, Franklin Institute Monograph No. 3, pp. 87–102. (For further information, see Luis Trabb Pardo and Donald E. Knuth, "The Early Development of Programming Languages," in N. Metropolis et al. (eds.), *A History of Computing in the Twentieth Century,* New York, Academic Press, 1980, pp. 197–273.)

Poley, Stan. 1957. "SOAP II Programmer's Reference Manual." IBM Corporation, Form 32-7646, 94 pp.

Poley, S., and G. Mitchell. November 1955. "Symbolic Optimum Assembly Programming (SOAP)." *650 Programming Bulletin 1,* IBM Corporation, Form 22-6285-1, 4 pp.

Wolontis, V. M. March 1956. A complete floating-decimal interpretive system for the IBM 650 magnetic drum calculator. *Technical Newsletter No. 11,* IBM Applied Science Division, 35 pp.

APPENDIX

Number Perverter Demonstration Card (8/15/59)

Instructions for use:

Prepare console as follows:

Storage Entry	70 9000 9001+
Half Cycle	RUN
Address	8000
Control	ADDRESS STOP
Display	UPPER ACCUM

Place Perversion Card in read hopper.

Depress Computer Reset, Program Start.

Depress START and END OF FILE simultaneously on card reader.

The program should now be stopped with 8000 in the Address Lights.

Change storage entry switches to 60 8004 9001+.

Now the program is satisfactorily initialized.

Set 8004 to any number. Press Program Start. When the machine stops, the number will appear with its digits reading from right to left instead of from left to right.

You may reset 8004 and depress Program Start again as often as you wish.

The card:

```
40 9007 8000
20 9009 9002
65 8003 9003
14 9004 9005
00 0000 0010
10 9009 9006
50 1000 9000
19 9004 9001
```