1967
Turing
Award
Lecture

Computers Then and Now

MAURICE V. WILKES
Cambridge University
Cambridge, England

Reminiscences on the early developments leading to large-scale electronic computers show that it took much longer than was expected for the first of the more ambitious and fully engineered computers to be completed and prove themselves in practical operation. Comments on the present computer field assess the needs for future development.

I do not imagine that many of the Turing lecturers who will follow me will be people who were acquainted with Alan Turing. The work on computable numbers, for which he is famous, was published in 1936 before digital computers existed. Later he became one of the first of a distinguished succession of able mathematicians who have made contributions to the computer field. He was a colorful figure in the early days of digital computer development in England, and I would find it difficult to speak of that period without making some references to him.

Pioneering Days

An event of first importance in my life occurred in 1946, when I received a telegram inviting me to attend in the late summer of that year a course on computers at the Moore School of Electrical Engineer-

Presented at the ACM 20th Anniversary Conference, Washington, D.C., August 1967.

Author's present address: Olivetti Research Ltd., 4A Market Hill, Cambridge CB2 3NJ, England.

ing in Philadelphia. I was able to attend the latter part of the course, and a wonderful experience it was. No such course had ever been held before, and the achievements of the Moore School, and the other computer pioneers, were known to few. There were 28 students from 20 organizations. The principal instructors were John Mauchly and Presper Eckert. They were fresh from their triumph as designers of the ENIAC, which was the first electronic digital computer, although it did not work on the stored program principle. The scale of this machine would be impressive even today—it ran to over 18,000 vacuum tubes. Although the ENIAC was very successful—and very fast—for the computation of ballistic tables, which was the application for which the project was negotiated, it had severe limitations which greatly restricted its application as a general-purpose computing device. In the first place, the program was set up by means of plugs and sockets and switches, and it took a long time to change from one problem to another. In the second place, it had internal storage capacity for 20 numbers only. Eckert and Mauchly appreciated that the main problem was one of storage, and they proposed for future machines the use of ultrasonic delay lines. Instructions and numbers would be mixed in the same memory in the way to which we are now accustomed. Once the new principles were enunciated, it was seen that computers of greater power than the ENIAC could be built with one tenth the amount of equipment.

Von Neumann was, at that time, associated with the Moore School group in a consultative capacity, although I did not personally become acquainted with him until somewhat later. The computing field owes a very great debt to von Neumann. He appreciated at once the possibilities of what became known as logical design, and the potentialities implicit in the stored program principle. That von Neumann should bring his great prestige and influence to bear was important, since the new ideas were too revolutionary for some, and powerful voices were being raised to say that the ultrasonic memory would not be reliable enough, and that to mix instructions and numbers in the same memory was going against nature.

Subsequent developments have provided a decisive vindication of the principles taught by Eckert and Mauchly in 1946 to those of us who were fortunate enough to be in the course. There was, however, a difficult period in the early 1950s. The first operating stored-program computers were, naturally enough, laboratory models; they were not fully engineered and they by no means exploited the full capability of the technology of the time. It took much longer than people had expected for the first of the more ambitious and fully engineered computers to be completed and prove themselves in practical operation. In retrospect, the period seems a short one; at the time, it was a period of much heart searching and even recrimination.

I have often felt during the past year that we are going through a very similar phase in relation to time sharing. This is a development

carrying with it many far-reaching implications concerning the relationship of computers to individual users and to communities, and one that has stirred many people's imaginations. It is now several years since the pioneering systems were demonstrated. Once again, it is taking longer than people expected to pass from experimental systems to highly developed ones that fully exploit the technology that we have available. The result is a period of uncertainty and questioning that closely resembles the earlier period to which I referred. When it is all over, it will not take us long to forget the trials and tribulations that we are now going through.

In ultrasonic memories, it was customary to store up to 32 words end to end in the same delay line. The pulse rate was fairly high, but people were much worried about the time spent in waiting for the right word to come around. Most delay line computers were, therefore, designed so that, with the excercise of cunning, the programmer could place his instructions and numbers in the memory in such a way that the waiting time was minimized. Turing himself was a pioneer in this type of logical design. Similar methods were later applied to computers which used a magnetic drum as their memory and, altogether, the subject of *optimum coding*, as it was called, was a flourishing one. I felt that this kind of human ingenuity was misplaced as a long-term investment, since sooner or later we would have truly random-access memories. We therefore did not have anything to do with optimum coding in Cambridge.

Although a mathematician, Turing took quite an interest in the engineering side of computer design. There was some discussion in 1947 as to whether a cheaper substance than mercury could not be found for use as an ultrasonic delay medium. Turing's contribution to this discussion was to advocate the use of gin, which he said contained alcohol and water in just the right proportions to give a zero temperature coefficient of propagation velocity at room temperature.

A source of strength in the early days was that groups in various parts of the world were prepared to construct experimental computers without necessarily intending them to be the prototype for serial production. As a result, there became available a body of knowledge about what would work and what would not work, about what it was profitable to do and what it was not profitable to do. While looking around at the computers commercially available today, one cannot feel that all the lessons were learned, there is no doubt that this diversity of research in the early days has paid good dividends. It is, I think, important that we should have similar diversity today when we are learning how to construct large, multiple-access, multiprogrammed, multiprocessor computer *systems*. Instead of putting together components and vacuum tubes to make a computer, we have now to learn how to put together memory modules, processors, and peripheral devices to make a system. I hope that money will be

available to finance the construction of large systems intended for research only.

Much of the early engineering development of digital computers was done in universities. A few years ago, the view was commonly expressed that universities had played their part in computer design, and that the matter could now safely be left to industry. I do not think that it is necessary that work on computer design should go on in all universities, but I am glad that some have remained active in the field. Apart from the obvious functions of universities in spreading knowledge, and keeping in the public domain material that might otherwise be hidden, universities can make a special contribution by reason of their freedom from commercial considerations, including freedom from the need to follow the fashion.

Good Language and Bad

Gradually, controversies about the design of computers themselves died down and we all began to argue about the merits or demerits of sophisticated programming techniques; the battle for automatic programming or, as we should now say, for the use of higher level programming languages, had begun. I well remember taking part at one of the early ACM meetings—it must have been about 1953—in a debate on this subject. John Carr was also a speaker and he distinguished two groups of programmers; the first comprised the "primitives," who believed that all instructions should be written in octal, hexadecimal, or some similar form, and who had no time for what they called fancy schemes, while the second comprised the "space cadets," who saw themselves as the pioneers of a new age. I hastened to enroll myself as a space cadet, although I remember issuing a warning against relying on interpretive systems, for which there was then something of a vogue, rather than on compilers. (I do not think that the term compiler was then in general use, although it had in fact been introduced by Grace Hopper.)

The serious arguments advanced against automatic programming had to do with efficiency. Not only was the running time of a compiled program longer than that of a hand-coded program, but, what was then more serious, it needed more memory. In other words, one needed a bigger computer to do the same work. We all know that these arguments, although valid, have not proved decisive, and that people have found that it has paid them to make use of automatic programming. In fact, the spectacular expansion of the computing field during the last few years would otherwise have been impossible. We have now a very similar debate raging about time sharing, and the arguments being raised against it are very similar to those raised earlier against automatic programming. Here again, I am on the side of the space cadets, and I expect the debate to have a similar outcome.

Incidentally, I fear that in that automatic programming debate Turing would have been definitely on the side of the primitives. The programming system that he devised for the pioneering computer at Manchester University was bizarre in the extreme. He had a very nimble brain himself and saw no need to make concessions to those less well-endowed. I remember that he had decided—presumably because someone had shown him a train of pulses on an oscilloscope—that the proper way to write binary numbers was backwards, with the least significant digit on the left. He would, on occasion, carry this over into decimal notation. I well remember that once, during a lecture, when he was multiplying some decimal numbers together on the blackboard to illustrate a point about checking a program, we were all unable to follow his working until we realized that he had written the numbers backwards. I do not think that he was being funny, or trying to score off us; it was simply that he could not appreciate that a trivial matter of that kind could affect anybody's understanding one way or the other.

I believe that in twenty years people will look back on the period in which we are now living as one in which the principles underlying the design of programming languages were just beginning to be understood. I am sorry when I hear well-meaning people suggest that the time has come to standardize on one or two languages. We need temporary standards, it is true, to guide us on our way, but we must not expect to reach stability for some time yet.

The Higher Syntax

A notable achievement of the last few years has been to secure a much improved understanding of syntax and of syntax analysis. This has led to practical advances in compiler construction. An early achievement in this field, not adequately recognized at the time, was the Compiler-Compiler of Brooker and Morris.

People have now begun to realize that not all problems are linguistic in character, and that it is high time that we paid more attention to the way in which data are stored in the computer, that is, to data structures. In his Turing lecture given last year, Alan Perlis drew attention to this subject. At the present time, choosing a programming language is equivalent to choosing a data structure, and if that data structure does not fit the data you want to manipulate then it is too bad. It would, in a sense, be more logical first to choose a data structure appropriate to the problem and then look around for, or construct with a kit of tools provided, a language suitable for manipulating that data structure. People sometimes talk about high-level and low-level programming languages without defining very clearly what they mean. If a high-level programming language is one in which the data structure is fixed and unalterable, and a low-level language is one

in which there is some latitude in the choice of data structures, then I think we may see a swing toward low-level programming languages for some purposes.

I would, however, make this comment. In a high-level language, much of the syntax, and a large part of the compiler, are concerned with the mechanism of making declarations, the forming of compound statements out of simple statements, and with the machinery of conditional statements. All this is entirely independent of what the statements that really operate on the data do or what the data structure is like. We have, in fact, two languages, one inside the other; an outer language that is concerned with the flow of control, and an inner language which operates on the data. There might be a case for having a standard outer language—or a small number to choose from—and a number of inner languages which could be, as it were, plugged in. If necessary, in order to meet special circumstances, a new inner language could be constructed; when plugged in, it would benefit from the power provided by the outer language in the matter of organizing the flow of control. When I think of the number of special languages that we are beginning to require—for example, for real time control, computer graphics, the writing of operating systems, etc.,—the more it seems to me that we should adopt a system which would save us designing and learning to use a new outer language each time.

The fundamental importance of data structures may be illustrated by considering the problem of designing a single language that would be the preferred language either for a purely arithmetic job or for a job in symbol manipulation. Attempts to produce such a language have been disappointing. The difficulty is that the data structures required for efficient implementation in the two cases are entirely different. Perhaps we should recognize this difficulty as a fundamental one, and abandon the quest for an omnibus language which will be all things to all men.

There is one development in the software area which is, perhaps, not receiving the notice that it deserves. This is the increasing mobility of language systems from one computer to another. It has long been possible to secure this mobility by writing the system entirely in some high-level programming language in wide use such as ALGOL or FORTRAN. This method, however, forces the use of the data structures implicit in the host language and this imposes an obvious ceiling on efficiency.

In order that a system may be readily transferred from one computer to another, other than *via* a host language, the system must be written in the first place in machine-independent form. This would not be the place to go into the various techniques that are available for transferring a suitably constructed system. They include such devices as bootstrapping, and the use of primitives and macros. Frequently the operation of transfer involves doing some work on a computer on which

the system is already running. Harry Huskey did much early pioneer work in this subject with the NELIAC system.

There is reason to hope that the new-found mobility will extend itself to operating systems, or at least to substantial parts of them. Altogether, I feel that we are entering a new period in which the inconveniences of basic machine-code incompatibility will be less felt. The increasing use of internal filing systems in which information can be held within the system in alphanumeric, and hence in essentially machine-independent, form will accentuate the trend. Information so held can be transformed by algorithm to any other form in which it may be required. We must get used to regarding the machine-independent form as the basic one. We will then be quite happy to attach to our computer systems groups of devices that would now be regarded as fundamentally incompatible; in particular, I believe that in the large systems of the future the processors will not necessarily be all out of the same stable.

Design and Assembly

A feature of the last few years has been an intensive interest in computer graphics. I believe that we in the computer field have long been aware of the utility in appropriate circumstances of graphical means of communication with a computer, but I think that many of us were surprised by the appeal that the subject had to mechanical engineers. Engineers are used to communicating with each other by diagrams and sketches and, as soon as they saw diagrams being drawn on the face of a cathode-ray tube, many of them jumped to the conclusion that the whole problem of using a computer in engineering design had been solved. We, of course, know that this is far from being the case, and that much hard work will be necessary before the potential utility of displays can be realized. The initial reaction of engineers showed us, however, two things that we should not forget. One is that, in the judgment of design engineers, the ordinary means of communicating with a computer are entirely inadequate. The second is that graphical communication in some form or other is of vital importance in engineering as that subject is now conducted; we must either provide the capability in our computer systems, or take on the impossible task of training up a future race of engineers conditioned to think in a different way.

There are signs that the recent growth of interest in computer graphics is about to be followed by a corresponding growth of interest in the manipulation of objects by computers. Several projects in this area have been initiated. The driving force behind them is largely an interest in artificial intelligence. Both the tasks chosen and the programming strategy employed reflect this interest.

My own interest in the subject, however, is more practical. I believe that computer controlled mechanical devices have a great future in

factories and elsewhere. The production of engineering components has been automated to a remarkable extent, and the coming of numerically-controlled machine tools has enabled quite elaborate components to be produced automatically in relatively small batches. By contrast, much less progress has been made in automating the assembly of components to form complete articles.

The artificial intelligence approach may not be altogether the right one to make to the problem of designing automatic assembly devices. Animals and machines are constructed from entirely different materials and on quite different principles. When engineers have tried to draw inspiration from a study of the way animals work they have usually been misled; the history of early attempts to construct flying machines with flapping wings illustrates this very clearly. My own view is that we shall see, before very long, computer-controlled assembly belts with rows of automatic handling machines arranged alongside them, and controlled by the same computer system. I believe that these handling machines will resemble machine tools rather than fingers and thumbs, although they will be lighter in construction and will rely heavily on feedback from sensing elements of various kinds.

The Next Breakthrough

I suppose that we are all asking ourselves whether the computer as we now know it is here to stay, or whether there will be radical innovations. In considering this question, it is well to be clear exactly what we have achieved. Acceptance of the idea that a processor does one thing at a time—at any rate as the programmer sees it—made programming conceptually very simple, and paved the way for the layer upon layer of sophistication that we have seen develop. Having watched people try to program early computers in which multiplications and other operations went on in parallel, I believe that the importance of this principle can hardly be exaggerated. From the hardware point of view, the same principle led to the development of systems in which a high factor of hardware utilization could be maintained over a very wide range of problems, in other words to the development of computers that are truly general purpose. The ENIAC, by contrast, contained a great deal of hardware, some of it for computing and some of it for programming, and yet, on the average problem, only a fraction of this hardware was in use at any given time.

Revolutionary advances, if they come, must come by the exploitation of the high degree of parallelism that the use of integrated circuits will make possible. The problem is to secure a satisfactorily high factor of hardware utilization, since, without this, parallelism will not give us greater power. Highly parallel systems tend to be efficient only on the problems that the designer had in his mind; on other problems, the hardware utilization factor tends to fall to such an extent that

conventional computers are, in the long run, more efficient. I think that it is inevitable that in highly parallel systems we shall have to accept a greater degree of specialization towards particular problem areas than we are used to now. The absolute cost of integrated circuits is, of course, an important consideration, but it should be noted that a marked fall in cost would also benefit processors of conventional design.

One area in which I feel that we must pin our hopes on a high degree of parallelism is that of pattern recognition in two dimensions. Present-day computers are woefully inefficient in this area. I am not thinking only of such tasks as the recognition of written characters. Many problems in symbol manipulation have a large element of pattern recognition in them, a good example being syntax analysis. I would not exclude the possibility that there may be some big conceptual breakthrough in pattern recognition which will revolutionize the whole subject of computing.

Summary

I have ranged over the computer field from its early days to where we are now. I did not start quite at the beginning, since the first pioneers worked with mechanical and electromechanical devices, rather than with electronic devices. We owe them, however, a great debt, and their work can, I think, be studied with profit even now.

Surveying the shifts of interest among computer scientists and the ever-expanding family of those who depend on computers in their work, one cannot help being struck by the power of the computer to bind together, in a genuine community of interest, people whose motivations differ widely. It is to this that we owe the vitality and vigor of our Association. If ever a change of name is thought necessary, I hope that the words "computing machinery" or some universally recognized synonym will remain. For what keeps us together is not some abstraction, such as Turing machine, or information, but the actual hardware that we work with every day.

Categories and Subject Descriptors:

D.1.2 [**Software**]: Programming Techniques—*automatic programming*; I.2.1 [**Computing Methodologies**]: Artificial Intelligence; K.2 [**Computing Milieu**]: History of Computing—*people, systems*

General Terms:

Design, Languages

Additional Key Words and Phrases:

ENIAC, Moore School, optimum coding, ultrasonic delay line