CONSIDERATIONS FOR A COURSE FOR FUTURE-ORIENTED COMPUTER DESIGNERS

R. S. Barton

Rapporteurs: Mr. H. C. Lauer
Mr. J. Knight

Introduction

I want to discuss why we should want to teach anyone to become a computer designer, what sort of people we should teach and how. We are interested in the production of radical computer designers: people who, hopefully, will not have to serve long apprenticeships while they mature enough to take major decisions in their organisations; people who will think about technology and what can be done with it that is of interest to human beings. This is perhaps a romantic viewpoint but I think it is becoming more popular.

I am going to draw upon two sources. Firstly, the history as I have seen it, and secondly, the technology as it exists now and some conservative extrapolations of that technology. I have been around the field for a bit over twenty years and worked in some, though not all, parts of it. I thought a great deal about my own experiences and I have found out as much as I could about what really happened from people who were there. With respect to technology I have the advantage of not being enough of a technician to be overly taken by the wonders of the stuff, while having enough common sense to look at it as an intelligent layman might. I wonder what the componts that we have now are going to mean for the future.

Before I discuss computer design itself I should say something about my qualifications. I have not taught any computer design. I have taught a number of courses, never the same one twice, following topics in the ACM Curriculum such as 'data structures' and 'advanced programming'. I also have no particular qualifications as an engineer, although I have been associated in an intimate way with some machines. If I have to be labelled, one could say that I am a discontented programmer who, perhaps because of a lack of competence in dealing with complexity, set out to do something about it by changing machines. I did this first at a time when it was most unpopular to talk as if one could somehow combine the subjects of computer organization and programming.

Computer Science courses - Present and Future

Computer science courses at the university level have developed very hastily after the fact of the computer became obvious to everyone. It was at first exclusively and industrial development and it was only when there began to be thousands, even tens of thousands, of computers around, that the universities began reacting. There were of course a few pioneer institutions in England and the United States which got a very early start. Programs that are called computer science now were usually called something else in the early days. Among the earliest were the programs in applied mathematics at Harvard and the program at Manchester. By the middle sixties the pressure in the United States to have computer science courses was tremendous. It began to look as if every university and college was planning to have a program, far more than there were people qualified to staff them.

Starting a program then also meant acquiring equipment. That was a period of gifts and large discounts by the manufacturers but there was still the problem of raising funds. Building computers and computer design was not a fashionable subject for universities. In the fifties people with experience knew that it was not practical to build a general purpose machine, keep it running, and serve the problem solvers at the same time. The reaction was to break away from computer design altogether in the universities and to take a standard product from one of the manufacturers. This meant that during that period, IBM had a disproportionate influence and many schools acquired IBM equipment of the then popular vintages. Relatively few machines were built in the universities, one of the most interesting exceptions being the Rice University machine with which John Illiffe was associated.

Computer science courses usually started with departmental sponsorship, perhaps Electrical Engineering, perhaps Mathematics depending on the professor. If you look at the existing programs they tend to be strongly biased by the department and the people that initially set them up. You might find a department with twenty or so numerical analysts and automata theorists, in other cases you might find mainly electrical engineers. The latter were in a minority, I think, with the exception of schools like M.I.T., because of the unpopularity of building machines during that period. Then the subject of programming began to get just formal enough to make it possible to have some computer science groups based mainly on programming. This was the period of the obsession with syntax and syntax directed compilers. I shall have more to say about this later.

This meeting may be one of the first to address itself to the subject of producing a more balanced computer scientist. He does not seem to be referred to as an engineer in any of the papers I have seen at this meeting though I think it odd that we should think of the product of these courses as being a scientist. Some people prefer to think of him as an engineer because of the largely empirical nature of the subject. However, it does seem now to be desirable to put out a student who is not labelled as a circuit or logic designer on the one hand or a programmer on the other but who is more versatile.

In order to do this we ought to know what we want to produce, i.e., what the people whom we teach should become. It is not adequate to say that the output of our institution is to be a system programer who typically may work for, say, ICL. I believe we cannot talk that way because I am not convinced that there are going to be any system programmers in another five years, or ten years at the most. Similarly, we cannot describe our output as a man who is going to work for, say, IBM as a computer designer designer because the only sorts of positions that are available are specialist positions. There a man works as a cog in a very large machine and by the time that machine has gone through its appointed cycles a sufficient number of times the man is reduced to 'cogdom' for ever. I would prefer to think of our course as training for saboteurs.

Consider where the bright young man goes who has ideas about computer structure and who would like to design his own machine. Generally, he cannot go to the computer manufacturers because they are merely copying one another. Some have fairly large research establishments but the output of those establishments is usually ignored. One thing he can do is to start up his own company. If we were producing computer designers who are going to do that sort of thing we ought to teach them something about raising money, marketing, and so on; we would need to produce the complete man. That is not such a bad idea. We are certainly not in the trade school business and we should not be teaching the construction of the various manufacturer's software or the details of their equipment.

Computer designers ought to be generalists, which means that they cannot know very much about any given thing. For example, consider what it would do to a man's potential for further thought if he were required to master the structure of some large, popular operating system, in view of the lack of intellectual content represented in such a program. How much better for him to actually build some sort of operating system on a much more modest scale. As for hardware design, one can argue in a similar way. The really important thing is to get the potential computer designers to actually design something and build it.

That brings up the question of whether it is practical or not, i.e. can you afford to do it. We can afford to do it if we stop buying computers. One thing a computer science department does not need is a computer, as Howard Aitken concluded by about 1953. He knew at that time that a computer science group did not need one because it would get in the way of creative work. This does not mean that a department should not build a computer, or better yet, build many little ones, take them apart, and build some more. Opulence is bad for computer science, and I say that with some authority, being associated with a department which has choked on its own opulence. Big machines tend to dominate everything, both the organization and technical interest. The bigger the machine the worse it gets; it is actually going to control everything we do. Much of what can be learned should be learned by touch - by touching and kicking computers. With the money saved by not buying a big machine we can buy parts and use those parts to build things.

A course for computer designers ought to last for only one year, not be broken into elementary, intermediate and advanced topics. It should be mostly laboratory and should teach switching theory, logic design, computer organisation and programming all at the same time. Some people say that this cannot be done, that one must hold the machine still while programming it, then forget about programming while letting the machine vary. They would concentrate on simple applications of logic design and on applications of simple arithmetic circuits perhaps. Then, by some great vaulting motion they get into computer organisation. I disagree. I notice that one of the authors of the relatively few books on the subject, Gordon Bell, has cheated and used the historical approach by collecting together papers on the subject of computer organisation. I think that is quite good. It is in effect a course in pathology though not entirely because one can look for the key ideas that appeared along the way.

It is these key ideas which are important. Most of what else the students do should not be computer science. It is important that these students of computer design should not do too much computer science or be too specialised in any other area either. They could be electrical engineers in part, mathematicians in part, and other kinds of people in part. I suppose there will inevitably be specialist courses, such as artificial intelligence 1, 2 and 3, and courses in numerical analysis, but ideally there should be just one course which deals with the simple down to earth notions where we really know what we are talking about.

The course ought to be accessible to nearly everyone. It should be pretty much in the general education category, and that should be a guide to the content. If it is not worth teaching to the English majors, it may not be worth teaching to the future computer designers.

This is because the technology is so important that it must be part of the common knowledge or we will be overrun by it.

There is much about this course that is impractical, as experienced organisers of laboratories will know. I am advocating a kind of organised chaos with parts bins and equipment lying around where people are allowed to think before they are thought to be capable of thinking, where they express algorithms not in the form of Fortran programs but as hardware or combination of hardware and program. I am going to argue later about the implications of components, why this ought to be possible, and why it ought to work.

Let us consider what a computer designer is, and why anyone should want to design computers. There is one very great one, for instance, very nearly the complete engineer and incredibly talented. He is Seymour Cray, who designed the CDC and 7600. Recently he was quoted as saying that the 7600 is the last small machine he is going to work on. Therein lies his motivation. He wants to build a machine that is very big and very fast for a certain class of problems. I doubt if he cares at all what it is used for. He designs his super machine as a kind of art form. That is one kind of computer designer. Another kind belongs to the faceless design team of a typical manufacturer. It consists of battalions and divisions of computer designers, all optimising the living daylights out of something which is guaranteed to have absolutely no architectural integrity. The term 'computer architecture' was first used in connection with such a monumental system, and introduced I think, a serious misuse of the word 'architecture'.

To understand the term architecture, forget about the computers for the moment and think about buildings. An architect is an artist. He has some concept of what something should be like as a whole, how it should all fit together, how it should all be one thing. He is normally not able to build all of his building personally. He has to deal with workmen and supervise them. He has a communication problem, and if he is going to be successful he must use a great deal of energy in monitoring the work to see that his concepts do not get degraded in the execution.

A computer system must have an architect if it is to deserve the term "system". Someone should have some overall idea of what it is about. Unfortunately, I do not think there are many examples of computers which fulfill this consideration. They have generally been built as perturbations of designs of previous computers. The word "compatibility" is often used to suggest the general constraints which exist.

Now consider how one becomes an architect. People do not become architects by first being specialists in many fields; they do so because they cannot help it. Computer designers, using the term broadly, should have something in common with architects. Architects do not forget that buildings exist for people. Some may need to impress people from a distance with a monumental architecture and because of this less attention is paid to making people comfortable inside the buildings, but as Frank Lloyd Wright may have said, the greatest architect will be the man who builds a building that no one notices. The digital computer technology can only be justified, a truism but we forget it, by the utility of these things for people. It seems clear and obvious to me that the computer field has reached its present state because of two long standing notions that are not very tenable. The first notion is that computers are algorithmic processors of some sort which eliminate the human being by turning human activities into algorithms. This is the way people thought about using computers for applications in business. i.e. eliminate the clerk. The second viewpoint which was never as widely held but has had considerable influence is the artificial intelligence viewpoint. They seemed to argue that while you may not be able to algorithmatize the human activity but you might be able to design a program which could learn how to do what man does. In both cases the idea was to get rid of people. As a result, we have produced computer designers and system programmers who are feeding on their own subject matter. They tend to design something which is faster, bigger, or more automatic than was previously available and they completely forget that the thing makes no sense at all unless people are involved.

There is another viewpoint which is not yet popularly held but I think will take over in time. It will take over only because the people in the computer field are going to have to adopt it in order to survive. It is the viewpoint of the computer as a communications medium. This forces one always to think of people-to-people communication via the machine. It is a medium which adds to the usual capabilities of communications devices the possibility of storage and the possibility of processing the information or permitting interaction between messages. There is no point in trying to produce computer designers unless that kind of thing is well understood. Consider what they are going to design. It is not sufficient to produce a man who can go out and be happy building yet a faster multiplier. We have a sleeping technology that is quite remarkable in terms of some of the raw measures but is not at all remarkable in terms of its true effect and its actual application. It is a big hoax in these respects.

My iconoclastic computer designers will have to understand this kind of thing to begin with. They must to decide what they want to do with the technology as individuals. They must not think that they are going to prepare themselves for jobs with the Sperry Rand Corporation, or are going to be a computer designers or system programmers because it is one of the higher paid occupations. They must be people who think about the human activities that the technology can be used for.

Let us take the application to education as an example. During the sixties there was a flurry of interest in computer aided instruction. Computer companies quickly looked around for publishers or education related firms with which to merge and the marketing executives of the computer companies found themselves thinking about selling computers to schools. Well, fortunately, this never came to much and the question-and-answer box approach failed because it was totally uneconomic. The educators may not have realised that all of expensive electronic gadgets were not doing much good. Those machines were designed to get rid of teachers, not to improve communication.

But it is obviously a communications application: teachers to students, students to teachers, students to students. That is not the way it was approached It was approached as yet another opportunity to create suitable algorithms, optimise programs and optimize the form of instruction. The very idea that everyone would learn in the same way and that there is an optimal method for all of us in absurd. There was a potential communications application but nobody took the time to think about what machines to really do something in education might be like. So, I think that education is a field still wide open and I would like my computer designers to be interested in that kind of problem.

Schools are space and time frames. There are rooms and one has to decide how big the rooms are going to be. In addition there is a system of dividing up the day and year into pieces and the learning process is then forced into this space and time frame. People have repeatedly commented that, in principle, computer technology ought to make it possible to get away from that. They really should have said communication technology aided by what we think of as computer technology. But we go on spending untold millions on buildings which are already obsolete. They have been obsolete for a decade or more and we are not spending much of anything - if anything - on looking at the way to use the new digital and communication technology to get away from those rigid space and time frames.

My first principle in teaching computer design is that people must be made to think about the social worth of what it is that they are going to do. They have got to discover it for themselves and that is difficult, particularly with engineers. They must get the point of view of the other culture. They must start believing that people are important. That is why they should not specialise in computer science too soon, but should look at the world through the eyes of the artist and the writer. They should be learning to cope with the technology in a way that removes the fear element.

A Historical View of Computer Design

In order to develop such a course we must understand what computer science is now and how it got that way. Let us start at the beginning and consider the extraordinary influence that Turing seems to have had. We can say that Turing established the notion that one could have a general purpose computer. Many people who have no familirity with the details of Turing's work were exposed to this idea of the general purpose computer very early and believed in it uncritically. Computers have a remarkable property that probably does not compare with anything that technology has produced before. One can change the function of a computer by changing its program from outside; i.e. the structure of the machine is, in part, soft. But the possibility of designing and building a "general purpose machine" is a different proposition altogether. A great deal of design effort has been severely handicapped because of this idea. I do not think that one can design a general purpose machine, as such. Machines have been designed that are called general purpose, but they are actually special purpose computers with purposes that are not necessarily clear to the designers. Thus, we have a whole group of artificial problems that need redefinition.

Some of the pioneer computer builders who not only thought of computers but actually built them and used them knew exactly what they were doing. For example, Aiken and Mauchley both designed computers because they had particular kinds of numerical problems in mind. I am sure that they became entranced with the machines themselves in the process, but they related the things that they were about to build quite directly to the models they had of the computations that they wished to perform. Both of those men were very practical indeed. They were building numerical engines, and they had straight-forward ideas of the computer structure suitable to that task. One of the first things that today's students of computer science should do is look at the work of some of these pioneers, i.e. Aiken, Mauchley and Eckert, Stibitz, Wilkes and Wheeler. All these people had problems to solve and they did not think of machines, except perhaps when they let their imaginations go on an excursion, as being very different from numerical engines.

Turing was different. He seemed to influence a lot of people to follow a more ephemeral path. For example, the artificial intelligence people can trace their origins to Turing's ideas. By the time I first came to the computer field, the general-purpose idea was very well established, even though it was quite a while before I heard of Turing. A characteristic of designing a "general purpose" computer is that there are no constraints. But a designer must have constraints in order to make decisions about what he is going to do as well as what he is going to do. For example, many of the pioneers were believers in decimal arithemetic. They used decimal arithmetic themselves, like all other people, and they thought it was worthwhile to build machines that would operate in decimal. But another school of design argued for the circuit efficiency of binary and established different constraints for its successors.

A second factor in the history of computer design has been the obsession with speed. This was partly the result of the very rapid development of electronic technology, which developed so rapidly that it was always possible to see very large factors of increase in machine speed in the near future. It resulted in a rapid rate of obsolescence of machines during the fifties and sixties which is only now just beginning to stop. Another factor contributing to the obsession was the parallelism versus serialism arguments. One outcome of an early meeting at the Moore School of Engineering has been that most of the major machines that have been built and widely used have had a parallel structure. The only reason for this had to be the belief "the faster, the better". The whole criterion of speed in design was divorced at a very early point from application. It was not a matter of building things fast enough. The things had to be fast in an absolute way, and each improvement in circuit speed and component speeds resulted in a new round of machine design.

Only occasionally was the goal of speed challenged. People would point to scientific and military problems which required it. Yet, in all of the machines that were built, including the majority of those built for business applications, the speed of the arithmetic unit was a major consideration. People working on these machines could not relate actual applicational needs to the machine specifications. Speed was an end sought in itself even as it has been in aircraft design.

The quest for speed and the resulting decisions in favour of parallelism have been taken to great extremes as in the large array machines like Illiac IV. It is a natural thing for mega-technology to be concerned with. Along with this obsession with speed was an obsession with size and sheer scale. Of course, in the beginning one had to have a certain investment in equipment in order to build anything at all; so there was a smallest machine that could be built with a given technology. The arguments in favour of large computers tended to emphasise the economies of scale and the increase in the number of possible states of the machines, but they neglected the considerations that some parts have low duty factors and some state transitions occur very rarely. What is the right size for a machine at any given state of the technology.

So today, we have the computer designers who believe the ideas that they can build a general purpose machine to do anything well, and that the faster and bigger a machine is the better. I do not think we can design sensible machines on that basis. However, we have been trying to do it; we have been building big, fast machines which have a short life and which are rarely missed when they are scrapped.

Another historical consideration is the accident that the principal forces in the computer industry tended to be the business machine manufacturers, i.e. their previous specialization was in making accounting machines, ledger handling equipment, punched card equipment, adding machines, etc. Businesses have used such machines for over one hundred years. The kind of people who think about business machines are generally accountants and other clerical people who are associated with keeping financial data. A characteristic of all of these machines is that they deal with paper. There are many wonderful horror stories about how the computer was made into a business machine. We certainly are aware of the impact of IBM's enormous and successful punched card business on computers. It is important to point out that there is now an essential absurdity about hanging business machines on electronic computers. Business machines utilise a highly developed, electro-mechanical paper handling technology. With the early computers it was far better to have a one hundred card per minute reader and a 150 line per minute printer than to have a ten character per second paper tape reader or a typewriter. But it is also true from the beginning that there was a considerable lag between the electronic speed of the machine and the input/output equipment. So the business machine manufacturers, having skills in this electro-mechanical area, devoted enormous effort to building machines that would do the same kind of thing, namely move paper around faster.

The problem with paper is that the cycle time for information on it is entirely different from that for electronically stored data. When we need the speed of electronics, we need it to deal with complexity and rapid change. Every time we use paper we are dealing with something that is relatively static. If anything is printed it is presumably worth keeping for a while. However, if you produce much paper in the modern information system, you are either admitting that you don't really need the system or you are producing errors at a very rapid rate. Once paper is produced and distributed, then you have all the problem of changes and that means more paper, and so on. The contradiction is obvious between the time honoured clerical methods of gathering data and producing hard copy, and at the same time insisting upon a constant improvement in speed and ability to handle complexity in the electronic part of the machine. Of course, we are moving in the right direction with time-sharing, online entry of data and program, cathode ray terminals, etc., but this is not the main thrust; it is a secondary thing and it is obscured by our lack of understanding of what these machines should be, i.e. whether they are to be properly regarded as communication devices rather than as processors.

We should be in the twilight of the era of paper technology.

I remember one of the early revelations of some of my colleagues about twenty years ago of the difficulty of programming a sort on one of the first of the large stored program machines. It could hardly sort records faster than a punched card sorter would. They did not dream that that would be the case when they started to write the program. Of course, we have long since had very fast sort programs, but the very fact that anyone uses sort programs at all is anamalous. It is due to another attitude resulting from the dominance of the electro-mechanical business machine manufacturers and their stress on magnetic tapes, which are only a little better than paper-oriented devices. The industry in the early sixties and almost forgotten how to make drums, and tended to de-emphasize disks since marketing policies discouraged the use of devices of that sort. System programmers, not knowing any better, had wasted much effort basing systems on tape drives. Tapes have the interesting property that they guarantee that most of the information stored on them is relatively inaccessible in comparison with what would be the case if it were stored elsewhere. In some cases, the data would be better kept on paper in filing cabinets.

All of these things in which we have believed - the ideas of the big, fast, general purpose machine driving paper-oriented input/output gear - have conditioned the way we design computers. We have evolved, on one hand, from people who built special purpose machines to solve numerical problems and, on the other hand, from people who wanted to do with computers what they did with paper-oriented business machines. Instead of successfully designing general purpose machines all we have really done is perturb designs which were not general purpose at all.

These attitudes created, in part at least, the gap between system programmers and computer designers. That gap is inevitable because people accept uncritically and conventional orthodoxies that happen to be current. They do not think about what they are doing. System programmers have evolved in a way parallel to computer designers with an equally misguided set of notions.

There is no doubt that the first electronic computers were very difficult to build. Some important design decisions had to be made just to get the things to work for even a few minutes at a time. There were not very many of them for a while, and because of that the attitude was quickly established that the time of machines was very valuable, more valuable, more valuable than the time of people. In response to a perceptive question from a person who recognized that programming was a difficult, demanding task, Von Neumann dismissed it as work to be assigned to graduate students.

In fact, in the early meetings, an idea that Von Neumann put forth was that one of the ways to tell whether a problem was suitable for a computer was to count the number of multiplications - it was necessary to have a lot of them. This was a common attitude then. The only dissent was in some very early remarks by Stibitz in about 1948 at one of the first Harvard Symposia. He daringly suggested that there might be some value in doing small problems with a computer, and that perhaps one might build into machines the facility of handle mathematical notation (for instance parentheses and subscripts). He suggested that it would be easy to do. People could not accept this because of their view that it took a lot of multiplications to make a computer problem. From this, it naturally followed that the machine had to be fast at multiplication. It also followed, by another line of reasoning that it was worthwhile taking any amount of pains to program the thing.

The result is that the engineers had to build machines with fast multipliers and it was silly to question how easy it was to program. The programmers were content for a while to write programs in absolute - a task made more difficult by binary machines - but they found that they were not very good clerks, and they started developing programming aids. In the early days, the aids for machines like the IBM 701 and 702 were primitive. One did assemblies using card sorters and reproducing punches, obvioulsy because machines were considered too valuable to use for that kind of low level activity which had no multiplications.

This attitude about programming aids accounts for much of what has happened since. For example, the development of FORTRAN, which had a pronounced effect on programming, was conditioned by the feeling that code produced by a translator must compare favourable with the hand-tailored product of the hypothetical expert programmer. This was more important than anything else, and it has influences all programming system design since. There were a few embarrassments, of course. It was very difficult at first to build compilers that would compile in reasonable amounts of time, but that was overlooked for a while. This established the precedent of making compilers and operating systems the number one application of computers.

After several years it was noticed that small runs predominated in computer usage. Then the designs of compilers were changed to favour the actual preparation of source programs instead of object program performance. There was also some effect on the design of operating systems to favour the small users, though the large machines quickly became inaccessible to most of their potential users. The large machine is surrounded by an organisation and staff dedicated to it. These people can talk to one another about secondary and tertiary problems having to do with the machine, and easily forget that the machine can actually be used for something. Like the designers of the same "general purpose" machine, they are divorced from application.

The difficulty of communication between system programmers and designers is, in effect, a phony problem. They have nothing to say to each other. Neither is concerned with what the machines are for. They are interested in them only from their own point of view based on their own conventional orthodoxies. We must begin again by teaching people what design is about, by having them design some simple things first, by having them build the machines they program. We must not produce specialists, but generalists - people who can take into account the human element and what things are for.

If we succeed in teaching people this way, they ought to be able to design things other than computers. They might design, for instance, bathrooms and avoid the awkward arrangement of a sink with two separate, spring faucets for hot and cold water. They could see in proper perspective the design of an on-line editing system which was the subject of a well known research activity which I visited recently. It was an introspective activity in which, it turns out, the principal use of the system is for the design, preparation, and editing of its own program, with little concern for such human factors as the tilt of the CRT screen.

To teach students who can carry on in the present way we can keep them working on real problems. This is often the case in present computer science programs. In the last few years system programming people have developed a mystique and much lore about syntax and syntax directed compilers. There are several courses available in each of many schools on this subject; and by now, many people know a lot about it but not what to do with it. There is something very sterile about a compiler generating a compiler and about programs written to check syntax. But no one uses syntax as a partial definition of a machine, for example, to make it self-explanatory in some particular sense.

We ought to treat compilers - and operating systems - as being mainly of historical interest. In the early fifties interpretive programs were used but these were slow and they had a lot of overhead, so people started writing compilers. This started a "school of static programming". It encouraged lack of generality by putting constraints on the programmer that gave him an artificial frame into which he had to fit. Furthermore, it encourages the artificiality which results from the design engineers who are primarily concerned with the fast multiplier.

Alternatively we could have taken the attitude that machines were wrong and that we ought to be able to run interpreters efficiently on them. For example, consider the evidence that has been accumulated by the APL system. How can people who on the one hand are constantly talking about the importance of rapid matrix operations and on the other hand be quite happy with an interpretive programming system which encourages, in fact almost forces, the use of matrix constructs. Users are fooling themselves. The APL machine is a very very slow machine by present standards, yet it satisfies many people because it is the only machine they have had to use that will respond in a reasonable way. Of course, you could build a very fast APL machine. It is a sufficiently constrained task to build a machine which is itself an APL interpreter. It is this point of view which we must teach the student: that it is perfectly reasonable to tackle the problem of building a machine in the broadest sense.

You can teach them using Gordon Bell's building blocks, Charles Seitz's building blocks, your own building blocks. It is a natural thing for computer scientists to think about such machines and to blur the distinction between designing the hard part and the soft part.

Technology and Design Goals

What we can teach and how we can teach it depends a lot upon the state of technology today and upon the direction it is heading. I want to make some comments based on my extrapolation into the future. These do not go very far ahead but I think they are all inevitable. They have nothing to do with the wonders of yet faster components to be expected but they do have something to do perhaps with the possibility of them being less expensive. I'll also try to tell you something about my own design philosophies and notions.

Consider which technological facts are going to be important. Charles Seitz has touched on the main one, and that is the sudden availability, after many years of promises, of integrated circuit storage. About two years ago, Rex Rice, one of the workers in the semiconductor industry, had just given a talk on fabrication of machines with new components having been engaged himself for several years in building a very elaborate machine. He reached the point where he began to realize that he had started his elaborate machine to soon, so he said to us that the important thing to system designers right now is that the cost of storage (per bit) is going to be roughly proportionate to size for fairly small stores, from say a thousand bits on up. At the time, I took note of what he said but it took me a long time to understand why that should be important. It is now true that bipolar and MOS integrated circuit chips are available as storage devices of various sizes, ranging from, say, sixty-four bits to 256 bits, 1024 bits and larger. All the component manufacturers are engaging in the usual battle of promises of lower prices and larger sizes, while computer manufacturers are concerned with the possibility of replacing their core stores, and actually doing it in some cases, with these new stores. It's an evolutionary thing from their point of view. They are going to plug in semiconductor storage in place of the old core storage. Of course, with core storage, one had to have a fairly large size unit to get to the point where the cost per bit was reasonable and so with this experience leading the way one finds discussions of monolithic assemblies of what initially comes from the manufacturers as small memory chips.

Now suppose we look at the whole thing again from the start, and what we see is the possibility of putting storage in the machine wherever we want it, not just in one large unit. The classical system diagram which shows control, arithmetic, storage, input, output would fragment right away into a very complex (and useless) picture with a lot of storage, a lot of control, perhaps a lot of arithmetic, and (along the same lines I mentioned previously) lots of input and output based on terminals, in order to get at people and to allow communication between people.

Moreover, as Charles Seitz has mentioned, in many cases where one used hard combinational logic, one can instead store the equivalent of the state table or a characteristic vector for a logic function. And so a good deal of what was thought of in the older systems as control now slips back into the province of the programmer but at a level where many programmers for a long time have not seen fit to work. It is at a level where representation is important, where one has to think of the codes that are suitable in the particular position you are using. This means getting away from the notion of a general-purpose anything and away from the idea of universal high-level languages, for instance, without requiring the designer to work in a way that many microprogram designers now work. It seems to be true that much microprogramming is done by people who ordinarily do logic design, because typical system programmers are not prepared to think at that level of detail, timing, etc.

Because of the economic factors involved, there are just two kinds of things that get into large scale integrated circuits in vast quantities: the ubiquitous electronic desk calculator (which is produced at a rate of a million per year now!) and location-addressed storage. For the latter the semiconductor firms are competing in the time-honoured fashion by trying to drive each other out of business and by trying to find things that they can sell in sufficient quantities where the price will hold up.

Now we can make an interesting observation. Suppose we could buy field programmable read-only memories and, as a byproduct of testing them make most of the combinational logic circuits that you might need to use simply by fusing the appropriate links. This will revolutionize design practise. There has been some discussion in this seminar of the implications of the classical method of designing from state machines, generating some combinational logic, doing some minimization. Finally, when someone makes a change, you start all over again. Much of that process disappears if we can store the state machine. Then when one does make a mistake, there is very little involved in making some new chips. I.e., the design process can become a lot softer and one can put small amounts of semiconductor storage wherever it is needed. This one result of the technology is the most important thing affecting the way we teach and the way we do computer design. It is not merely something that can be faster and cheaper, but something which changes the whole nature of design.

It is because of this that the introductory course in computer science ought to, from the very beginning, combine logic design (in simplified form) with the ideas of programming. There should be very strong emphasis on programming where choice of the representation is free. This is in contrast to the programming languages of the last ten or fifteen years where choice of representation is very restricted. They impose particular ways of handling characters. Fixed sizes of numbers, fixed conventions on floating point numbers, and until very recent times extremely limited data structures. It took a long time for vectors and arrays to get into programming languages in anything like a general fashion and tree structures are still not quite there.

Programmers do not appreciate the idea that representation is where everything starts and that the representation you choose should depend entirely on what you want to do with it. There is no best representation, so if machines become mostly storage, then the choice of representation becomes much more important than before.

I think that above all, the most general notion that has been put forward in any form in computing is the notion of hierarchical structure. What is interesting about the Algol language and a reason why it has considerable theoretical impact is the notion of recursive definition. More recently, Dijkstra has advocated a programming style which really amounts to confining oneself to hierarchical structure and to doing away with such ad hoc constructs as branches. We might consider the state machine model as one which is usually thought of as a single level construct and extend it to a hierarchy of state machines. Wherever one can extend the definition by making it recursive, one gets generality almost for free. It is the only way I know of that this can be done, but so far it has not influenced machine design very much. There are a number of instances of machines that have better capabilities for the nesting construct than others, but the vast majority of machines are still single level machines. In fact, the vast majority of machine design decisions seem to be taken about choices between parallel and serial structure than anything else. The things that computer design should be concerned with have to do with representation and concurrency. (I might mention in passing that semiconductor storage is inherently serial because of its packaging limitations; thus the old design constraints which occupy logic designers need no longer apply).

If we really believe that we can have a usable, general-purpose machine, then we must recognize that the kinds of things we want to represent are not known in advance. We cannot make any decisions, therefore, about the matters which notations like Bell's or the APL notation deal with very well. The questions of representint parallelism, word structure, how the instruction is divided up into fields, and that kind of descriptive material, is very well handled by those notations. But if we need a machine which will let us use <u>any</u> notation which is appropriate for expressing the program and any representation of the data and program which is appropriate to the frequency of use of the parts and the likelihood of change, then what can we say about the machine? The answer is that we cannot make any structuring decisions about word size, parallelism, etc., in advance. Instead, we would need to specify the structure of the machine as part of what is normally thought of as a program.

List processing was mentioned as an example of something that seems not to run very well on some super-computers. This is not a criticism of the idea of list processing, which after all is intended to deal with a variable structure in one particular way. It is merely an example of my contention that these machines are all special purpose. We cannot claim that they are general purpose at all, and the fact that we can simulate something given sufficient time is beside the point if that time is not reasonable.

The main areas where speed is really required are the special-purpose situations. They might be dealt with not by going parallel (i.e. with large, synchronized transfers of bits) but by going serially into individual, decentralized components of machines, and arranging the machine as a whole so that very large numbers of these components can work concurrently (only loosely synchronized). This changes the interconnection problem to one which begins to look more like those of telephone switching systems.

Periodicity is another aspect of computer design which we must consider. A peculiar attitude we have had for some time is exemplified by relief from not having to do minimal latency programming of the drum computers and, at the same time, annoyance that we do not have better disk systems. Programmers have been reluctant to tackle the problem of storage hierarchies and the problems of integrating periodic storage into a system. The reasoning is usually that we can expect very large, fast, cheap random access store in the near future, so there is no need to worry about these problems now. Noteworthy exceptions are the Atlas paging scheme and more recently, segmentation mechanisms and access queuing on a few other machines. But rotating storage devices are generally regarded as inconvenient things, and consequently second class designers are assigned to them.

Periodicity is something that is going to be there all the time. It is going to keep coming back in one way or another so students should learn something about coping with it. For instance, one can certainly build a machine with a periodic store and not much of any other kind of store, which will out perform any other machine used today for business applications. There are no speical programming problems. This is also true of time-sharing. The STAR machine is another example of facing up to periodicity in the way that the modules of storage are made available one after another, and of course the 6600 did a similar thing with the I/O processors. As a systems idea, periodicity is important. It is not something that gets in the way, it is something that can be utilized, and there are some very appealing low cost periodic stores available now. In fact, we could use some of these new components to build functional simulations of the classical machines, like the EDSAC. This would really be an eyeopener to people because machines have not improved that much in organization since 1950.

We can consider some alternatives to the global design goals of speed and size. For example, there is the problem of minimal representation. My personal estimate is that what, say, the 360 typically does in a program ought to be done in about one fiftieth (1/50) of the space. We have to look at machine organization differently and we have to look at the representation of the program differently. There are a lot of ideas and methods that have been around for a long time and that can be looked at this way. For example, Huffman codes can be used in context, so that the representation of the program depends upon the frequency of the elements which appear either statically or dynamically. A second interesting design notion might be the minimum power criterior, but I am not nearly enough of an engineer to know where that idea might lead. The idea is to see if you can get a thing done within some time constraint for the least expenditure of power. For example, MOS shift registers and stores can burn up a lot of power, so we ought to run them just fast enough to get your job done, not as a whole, but individually.

Representation and the Design Process

We have not had time to write any textbooks and I am not sure that computer design is ready for that kind of standardization. I urge the people with whom I come in contact to read things which are not normally found in the engineering curriculum. I am pleased that I have some support in this area from people who are much more formal and scholarly than I am particularly Edsger Dijkstra. A valuable reading is The Act of Creation by Arthur Koestler. It starts out with a lengthy discussion of nearly 100 pages on the subject of humour. Koestler propounds the theory that creative ideas are not the product of logical, systematic thinking at all, but are often first expressed as a joke. We have all the experiences of saying something, almost as a release from tension, in the form of a joke, and then saying "Hey, that's not a bad idea. It's worth following up."

Designers must learn how to think 'illogically' and must not be afraid of inconsistencies. While something new is developing in the mind, the inconsistencies are so frequent that if one stops to catch even a fraction of them, the whole train of thought dissolves. We must not let mathematics - or notation - get in the way of clear thinking. We must get across to people who, because they are human beings are going to be designers, the notion that they cannot depend upon any one piece of machinery to do it. Machines help, but every time one adopts a 'machine' - machine in the sense of a system, an algorithm or a standard method he gives up something and becomes imprisoned by it. For example, it was not until the late fifties that programmers even used graphs (in the topological sense) for any purposes other than sharing flow of control. Graphs, of all the general sorts of representations available, are probably the least confining. We do become trapped by notations and so must be prepared to invent them and discard them. That is something that I noticed that Don Knuth, in his remarks here last year, commented on. No schools, as far as he knew, ever had courses in "mathematical notation", either studying it, in the way one might think of comparative linguistics, or inventing it, or even philosophising about it. The more we formalize some of our educational processes, the more we tend to kill in the people subjected to these processes, the ability to invent ad hoc the tools that we need.

About ten years ago, I followe the particular track of using programming languages, particularly Algol and Algol-style languages as a model for a machine, and I found this quite fruitful. That particular way of doing things became almost machine-like, and gradually as people got involved who had not been there at the beginning and as the pioneers left, the 'machine' went on by itself. It was very difficult to introduce any new ideas in that situation. So if we produce students of design who are not prepared to cope with constant change as a normal thing, we are producing people of limited usefulness. Before they acquire the necessary maturity to participate in important decisions, they will be locked into some habitual way of thinking. It is not aging. It is a kind of human characteristic that has been accentuated over the last few hundred years. We do really want to find machines and methods to do things for us, but we take the life out of things this way.

Language, not programming language but our native tongue is also a handicap. I am very impressed with the thinking of William Huggins, a Professor at Johns Hopkins University. He talks about and illustrates his notions of iconic communication, that there are things which are best explained without using words using silent uncaptioned movies, tape TV or models, for example. I claim that most of the subject matter of the computing field can be better explained without the words that we use. Programming languages, some of the ideas of syntax, the ideas of translation, can be modelled with packs of cards, and illustrated succinctly without words. One comes closer to absorbing an abstract structure that way, or one learns to make his own descriptions from observing such a process. This is important for young people, particularly if we are concerned with universal knowledgeability in matters having to do with computers, but it is important also for the programmers and the design engineers.

In practice we cannot stop talking. What we must do is engage in the exercise of changing the notations and terminology frequently. Reformulating a difficult problem by just changing the words leads to a whole new context and sometimes a solution pops out.

Trying to pose design problems in unfamiliar context is helpful in unexpected ways. One of the more exciting examples of this is work that was done by some of our students in controlling an organ from a small computer. They designed and built what might be described as a modern version of a player piano. They did not really know what they were making at first and as yet it has not been used by composers, but I am sure it will be. They immediately learned some interesting things about programming. One of the things they did was make it possible to transcribe manually from standard musical notation to a linear notation to be used to drive the machine. One of the first things that they transcribed was a Haydn trumpet concerto (for organ); it was an extremely compact notation, actually more compact than standard musical notation. This constituted a program, run by an interpreter in the machine, cheating a little by changing the computer in a minor way. After all, they had a special problem. They just wanted to control the 80 odd keys and the various stops of an organ. If one wanted to justify such a frivolous effort without being defensive, we can think of it as an application in computers and control.

I like it for two reasons. First of all, knowing nothing at all about music myself, I am awed by the whole operation, and the other reason was that the thing was pleasingly slow, and small. It took approximately 3500 12 bit words to hold the interpreter and the music for the Haydn trumpet convinced as a result of observing that project that we are going to have a rash of digital musical instruments before very long. (I hope they are not made by a business machine manufacturer for I cannot imagine what would happen if they were - they might have punched card readers. This is perhaps my favourite example of a design project that was simple and put a lot of real constraints on the people involved. They did not have much money to spend and they had old parts that did not work very well; but after a year it was the only piece of equipment in the department that always worked.

Comments and Conclusions

Words are very important to the whole question of our course for computer designers. If we say that we have a programming laboratory which uses the tools of Charles Seitz or Gordon Bell, we would naturally say it is good experience for them. But if we call it a laboratory in logic design and machine organization, we are asked whether or not the students are learning enough about circuits.

I was asked if I wanted to produce students in my own image. I do not. I want to produce students who can discover who they are and can discover what they want to do. I belong to a radical school and am interested in the design of things other than computers. I hope the students will be interested in applying some ideas of information science in other areas. I can conclude by echoing a remark of U Thant about his fears for the future of a world run by computer technologists who care little about people. We have just barely enough time to start a counter movement.

Discussion

Professor Suchard opened the discussion by asking Professor Barton for his comments on the influence of micro-circuitry in design. He suggested that one of its most important elements was to free the computer designer from a fixed word structure. The restrictions of designing for ferrite cores, involving completely fixed word lengths, could give way to a variable format in which such factors could be decided independently for different parts of the machine - the channels could function by bits, other parts in octal, and so on. Professor Barton agreed, adding that all complex machines have had their instruction set strongly influenced by their word structure. The freedom of representation which becomes possible is an important opportunity presented by the advent of micro-circuitry and large scale integration.

Professor Page suggested that there were inherent dangers in the extent to which computer design was held in a straitjacket by manufacturers. He cited the danger in program design as in other research areas, of constructing experiments with regard to the methods of proof available. Professor Barton said that he thought that with increasing capacity, machines and devices would become more and more serial, with fewer inherent limitations. Dr. Horning questioned whether serial design was in fact less constraining than parallel to which Professor Barton replied that parallel was just a special case of concurrent, whereas serial involved only a single thread. Dr. Horning pressed the point about constraint and Professor Barton agreed that the serial approach was always appropriate unless the designer can show that the required speed can be obtained only with parallel logic. He emphasised again the importance of flexibility - for instance, in most comparisons, sums etc. the number of bits involved is very small; flexible design could give one the opportunity to do only what one needed to do, and the need for packing would be removed.

It was suggested by Professor Ercoli that the building of parallel memory devices was valuable, since serial memory could be simulated within it. Professor Barton countered that there was little point in building in parallel unless the device was inherently parallel, since parallel devices were by their nature inefficient in most circumstances.

The problem of looking for items of varying lengths in a list without looking through them all was raised by Professor Michaelson. This task was not easier serially, and he asked in what way serial design gave more freedom. Professor Barton replied that it enabled one to discard a pigeon-hole structure in memory - declaration could be used instead, and addresses could be computed. It was not enough to substitute serial for parallel memory; one had to go back to first design principles and get rid of addresses. This involved the design problems of a lifetime. He referred to a series of papers given at the Spring Joint Converence SYMBOL (Spring 1971, Rex Rice et al.) setting out principles not normally used in machine design, in which many functions of naming and storing variables and data were put in at the hardware level. Manufacturers' reaction was to say that programmers perform these functions already, at the software level.

Mr. O'Regan asked whether Professor Barton would recomment an efficient path for programmers. He replied by quoting Professor Perlis who summarised the essential in seven words: Definition, Evaluation, Substitution, Sequencing, Selection, Iterating, Binding.

Professor Randell mentioned the problem of the generality of recursion, and gave as an example the Burroughs 6700. When it attempted to get a word, it had to compute an address, and this was hidden from the user. This suggested an inherent recursion absent from Professor Bell's notational description. Professor Bell replied that the process described was obviously inherently recursive, but Professor Randell indicated that he was referring to recursiveness of description, not of action. Professor Bell replied that he was not sure and would have to check his notation - if it did not cater for the problem then it should.

The session concluded with Professor Page thanking Professor Barton and suggesting that in view of his departure before the plenary session next day, Professor Barton should really have been asked the questions he would most dislike having to answer.