DIGITAL SYSTEMS DESIGN

E. L. Glaser

Rapporteurs : Mr. J. G. Givens
Mr. N. Ghani
Mr. N. G. Kannellopoulos

Lecture 1 : Design of Digital Systems in the age of LSI

When Brian Randall first asked me to present these lectures, I felt quite competent to write on the subject and to be able to present them. Little did I guess at the agony that awaited me in the process of getting these few thoughts down on paper. The source of this discomfort has been the increasingly rapid change that is taking place in the field of digital circuitry today. During this last week, I found that I had to learn of three new processors that were variants on an existing one, plus two additional new types of memory and switching chips that have not been seen before. This is not an unusual week, merely the most recent example. It is like trying to play an athletic match with the rules changed during the contest. One thing that is sure, any design that uses present day chips will be obsolete long before its life cycle as a product is completed.

The next problem to confront me was how to talk about design and education. True, I have been a professor in the past, however, I am now what might be called an unfrocked professor. Still, I am finding more and more in my return to industry that the problems I faced in building laboratories within the academic environment are no different from those that I find facing me daily in industry. In both cases, it is necessary to structure laboratories that will be good for more than just one project, and are flexible enough to meet this ever increasing pace of change facing all of us.

For this first lecture, I should like to talk about the problems of digital systems design from the standpoint of circuits. Certainly, one of the most important circuit innovations to ever take place is the emergence of the microprocessor. Because of the importance of this innovation, I should like to devote the entire contents of my second lecture to the subject of this important design element. During the third lecture, I am going to attempt to pull together some of the aspects of systems design that have changed because of this changing technology, and at least pose those questions that I consider to be important. Unfortunately, solutions are few. Perhaps, you may have the answers. I can only say, I fervently hope so.

A few years ago, I was introduced by a friend to an old English proverb. At least he said it was an old English proverb. Namely, "An engineer is somebody who could do for a shilling what any fool can do for a pound." In short, the art of engineering is applying organised knowledge, science, to the solution of problems taking into account the necessary elements of time, money, resources, and suitability. Engineering is an art since it does depend on the value judgement and practice of the working engineer. Design is the culmination of the art of engineering in that it produces a coherent structure, be it electrical, physical, chemical, or logical, to accomplish a specific end. This structure must first have existed in the mind of the designer. All of these points are obvious. Yet, it does not hurt us to remind ourselves that these truisms are still fundamentally true.

In the practicing of the engineering art, one of the most critical factors must be the cost equation that either explicitly or implicitly comes to play during any design task. Optimisation is a concept and a term brought about primarily in the art of engineering. We find ourselves today, however, optimising systems often against outmoded concepts of cost and effectivity. I should like to give you a specific example that I encountered about a month and half ago. A particular project in our corporation was using one of the standard microprocessors. This particular product had been in development for sixteen months and it appeared as though it could be under development for another sixteen months. The problem was that they were trying to get a single processor to do many tasks. They were doing this in the classical method of computer scientists, namely, multiprogramming. The intellectual juggling act of multiprogramming is a very satisfying task, but it is not necessarily the best way of solving all problems in computing. In this specific case, only five units were to be built for a special application. The cost of design for each unit already was a factor of ten over the parts and labour cost of assembling each unit. It was found that by going from one to three processors tied together on a common bus, that the software could be finished in approximately six weeks, and the results were as predicted. Interestingly enough, the total cost of each unit from the standpoint of just parts alone dropped. Do you see why? It is because the amount of program required to run three processors in this environment was less than a half that required to run an individual one in a multiprogramming mode. Further, the programming could be much less abstruse and specific routines did not have to be written in several forms in order to optimise execution time. So, although three processors were being used, the cost was more than offset by cutting the amount of total ROM involved.

Compared to the earlier solution they were working on, multiprogramming was a good solution. It was simply the wrong solution for the time. This changing cost equation is one of the most fundamental problems that we are all facing today in our design activities.

One final point to be hammered home here. Three to four years ago, any designer proposing to produce a small desk-top machine based on a microprocessor would most likely be using a cassette as his low-cost backing store. The same designer working as long ago as two and a half years, and perhaps up to today, would be working with either a floppy-disk or a mini floppy-disk. What about the designer of today and next week, or next year? What happens with the emergence of bubbles? Besides, there is always CCD memory, with our old friend the cassette, as a means of a high-speed load and store. Admittedly, the CCD is not the same as a floppy, it is not removable, and it is volatile. However, by using it as a form of cache against a less desirable magnetic tape cartridge, the two requirements of removable non-volatile memory and random access memory have been divorced, and the same function is filled in a somewhat different method. This lesson is important. Often, we do not take full advantage of the new technology because we are too prone to try to fit the new components into an architectural and conceptual design that was based on a previously valid, but no longer effective, cost equation. Unfortunately, an optimum solution is in today's environment one of our most perishable commodities.

During the remainder of this lecture, I should like to discuss some of the problems that are new to the digital systems environment, as well as some old ones and a few that are still around but occasionally forgotten. First, how does one pick a particular component for a job? A good rule of thumb, I have found, is that if I can use a single-chip processor to do the job, I probably will. That does not mean that I am going to put in an 8080 instead of an AND gate. It does mean, however, I may put in an F8 or an 8048 to replace a large collection of logic. This class of chips is highly "plastic". The designer can form fit in any desired chip with whatever characteristic the designer wishes as long as the processor is fast enough to meet the design input/output requirements. In at least one case I am familiar with, a single chip processor replaced a servo system that was used to maintain constant speed on a disk drive. It is not that the chip processor was inherently cheaper than the discreet analog components that were used previously, rather, it is because the processor was more reliable and being one part instead of fifteen required less labour to assemble. Picking the right chip for design is again part of the art of engineering. Obviously, the chip has to be sufficient to perform the functions the designer requires. If the chip is more than sufficient and is economical, why not use it? If however, it will do only a small part of the job and there are many of the functions of the chip that are not usable, then it is highly desirable to look for other solutions. A chip that won't quite make the design requirements but that can be supplemented by other external circuitry, is a perfectly adequate solution. An example of picking the wrong chip for the job initially was a particular automaton design performed here in this country. A bit-slice chip was being employed for a particular function. After much designing, it proved to be totally unfeasible. The device needed no arithmetic and the chip was only being used for register storage and exclusive OR. Obviously, an MSI design was in order in this case.

The choice of circuit families is one that produces many arguments at all design meetings. Today, life is getting easier since most circuits are compatible with TTL levels even though they may not be true TTL internally. The key here is compatibility. It does not seem worthwhile to design a system which is rather small, must interface with other TTL circuitry, and yet for the sake of some other designer's whim be implemented entirely in ECL. Equivalently, if I am designing a very large system where speed is of the essence, ECL might be the best solution, interfacing with the TTL environment only when necessary. Let the application dictate the circuit family to be chosen. Although, if you have already a set of circuit cards in production and at hand that will do the job, then economics becomes a very strong factor.

In terms of details of circuiting design, it seems easier to express them as a set of random thoughts dressed up as though they were law. Let us call these the follow-on to Murphy's law, or all the things that the digital designer should know and do but has probably forgotten:

1. Cleanliness is next to godliness and this is particularly true for ground, power, clock, and signals.
2. A nanosecond is almost a zero but not quite, and the approximation falls down when the nanoseconds come in large numbers.
3. Although the design of a system may be digital, it is implemented in an analog world; and although <u>you</u> may forget, mother nature does not.
4. Unless you are interested in job security, remember that somebody else is going to have to maintain and modify what you design. Therefore, make it possible to be maintained by a highly-trained gorilla.

In conclusion, the process of design has become more interesting because we have more interesting components to deal with, but because of this, the challenge to fit them together has also increased. Optimisation is something to be striving for, however, let's make sure that we are optimising the right thing. A design that will cut 5% of the cost of a final item to be produced is of no interest if that additional engineering adds 10% to each item produced because of the large non recurring cost that must be spread out over the items that are produced. Remember, although something is designed once but is produced and maintained for the long term, unless the number being produced is large, the design <u>never</u> is free but is inherently costly.

<u>Discussion</u>

<u>Professor Hoare</u> began the discussion by suggesting that in order to solve a design problem, one should go to a place where design costs are negative (that is, a university). He felt that the things that should be taught are those aspects of computer science which are less ephemeral, such as algorithms, ideas of structure, and organisation. As an example, Euclid's algorithm will still be with us independent of past or future hardware developments.

<u>Professor McKeeman</u> asked about the problem of re-educating practising engineers in firms such as Dr. Glaser's. <u>Dr. Glaser</u> replied that although he tried the usual methods, for example, seminars, courses, etc., he felt that less than half of his present staff would make the transition to the new technology. These people would have to be put into an environment where they could still do useful work, well buffered from the new ideas. He pointed out that this was simply history repeating itself, in that the same problems appeared in the transition from relays and plugboards to flip-flops and stored-program logic.

Turning to Professor Hoare's comments, he agreed that non-ephemeral topics should be taught, but felt that people should be able to cope with all aspects of the design environment. For example, they should know how to produce a working design quickly, not just one that was fast or cheap.

Professor Page wondered whether Professor Hoare had over-simplified the situation in assuming that the primitives used in algorithms would remain constant. Dr. Glaser added that certain algorithms tend to be taken for granted, because certain primitives are taken for granted. For example, division was not a primitive at one time; now some people take transcendental functions as primitives. Professor Heath asked if this was the old argument of top-down versus bottom-up. He tended to advocate going from the middle outwards, while keeping an eye on both top and bottom.

Professor McKeeman countered by quoting a Polish proverb, "When crossing a swamp, keep one foot on solid ground," implying that by starting in the middle, one may have nothing to stand on.

Dr. Glaser : "Design is an art; part of it is taste, and we put into it as much rigour as we can." His definition of a good design was one that did what it was supposed to, at the right time and the right price.

Professor Randall quoted one of the designers of Colossus as saying "Any fool can build a bridge, it takes an engineer to build a bridge that can only just stand up."

Professor Vranesic turned the discussion back to teaching by asking about the problem of selecting devices for the shelves of a digital laboratory.

Professor Dijkstra suggested that such shelves should be kept empty and asked why such devices should be used at all in teaching.

Dr. Glaser's response was that he wanted to be reassured that someone could produce a design that would work. He would prefer to employ someone who had proved that he could produce working designs, rather than someone who had designed many things but had proven nothing. He used the analogy of trying to teach someone to swim without putting him in the water.

Professor McKeeman said that some of his students who went into industry complained that while the things they had been taught were of long-term value, they had learned nothing that could be used in the first year in industry.

Dr. Glaser pointed out that programming is design as well. He compared the idea of a laboratory with empty shelves to a computing science department with no computer, where people did not have to bother with the problem of getting programs to run. Professor Michaelson felt that proving a design with pencil and paper was fine, as long as the proof was total, which is difficult, and as long as the proof corresponded precisely to the real world, which is very difficult. Computer scientists have to be able to match up ill-defined user needs to partial theories, and this requires intuition which can only be gained though experience.

Lecture 2 : Enter the Microprocessor

The historian, George Santyana, once said that that nation which does not read history is doomed to repeat it. Today, a microprocessor seems to be repeating all of computer history known to date. True, the early ones were not quite back to Babbage although some appeared to be designed so that they could be powered by steam. It is both regrettable and lamentable that digital systems are being designed today in microprocessors, and yet no account is taken of software requirements. Forgetting some of the things that we discussed in the previous lecture, about the changing optimisation, the fact is that a $30,000 instruction program is still a large one. The fact that this program, consisting of individual instructions each of which is only a byte or two bytes, is housed in a very small area, namely a few ROM or PROM chips, does not in any way make it a simpler piece of software to produce. In fact, compared to many modern-day-large, and even medium and small computing systems, the microprocessor presents a primitive, if not positively savage, environment. Certainly, it cannot be described as friendly. In addition, system designers are finding, again, that it is not the arithmetic processing that is difficult, or even the normal flow, but rather the initialisation, the exceptions, and above all the input/output programming that is difficult. Should we really be that surprised?

For the purpose of discussion, it is useful to break the field of microprocessors into three classes.

1) the single chip processor, such as the Fairchild F8 or the Intel 8048.
2) the microprocessor based on a chip family such as the Intel 8080, Motorola 6800, Zilog Z80.
3) the bit slice system such as the AMD 2900 family.

It could be argued that there is a fourth class, namely, the small minicomputer such as the LSI-11. True, such systems do appear to have roughly the same capabilities as the upper end of the second family, however, this really begs the question. The Intel single-board computer is really equivalent to the same class of mini, and what we are designing is packaging. The question really comes to "Who is going to manufacture that particular assembly?" If the designer can deal with the individual chip, then there is more flexibility, but also more room for error and also more design work to be done. Certainly, in specific design, small minis should be considered as there is software that already runs on them, and is the major part of the design effort in many cases.

In the first lecture, we have already looked some at the use of the single-chip processor. These elements can be thought of as designer-specified, special purpose-chips. They can be used in a great many cases where a controller, a timer, or some other form of sequential logic is needed. Their versatility is only now being fully understood, and is yet to be fully exploited. The unusual design aspects they present to the system architecture are not significantly different from those presented by the more common microprocessors such as the Intel 8080. True, the single-chip processor may have a more bizarre structure, but other than that the problems are similar.

The question is, what to do about getting a microprocessor programmed? Here again, there are no neat, concise answers. The approach, too many times, has ended up being, "Put a large enough group of programmers on the job and you can get anything to work almost!" It is not clear what the "almost" modifies, the working or anything, or perhaps both. Most microprocessor manufacturers supply development systems. They are usually barely adequate to handle the design idiosyncrasies of the particular manufacturer's processor, and little else. We are beginning to see the emergence of new instruments from various instruments manufacturers aimed at this problem. Both Hewlett-Packard and Tektronixs are now offering systems that are meant to help in the programming of microprocessors. Unfortunately, although these systems are independent of a specific manufacturer's hardware, they also are independent of modern high level languages and some of the necessary aids. These deficiencies will probably be rectified in the future, and in fact, there is evidence that these corrections are already starting to take place.

Whatever has been stated about microprocessors goes even harder in the area of bit-slice architecture and true microcode. First, the bit slices require a much higher degree of sophistication in design. Second, in general, signals are higher speed and therefore more care has to be taken in the actual fabrication. Third, if a system is large enough, the actual system may consist of at least two and perhaps three or more somewhat independent processors. An example of this last structure would be using the arithmetic chips to form the actual central arithmetic element or mil. A microsequencer is used to drive the central unit and implement the details of the microcode. A third coupled system might employ a different type of bit slice for the macro instruction interpretation. There are no nice in circuit emulators to help check-out such a system. There are no really adequate languages in which to write the software. (In this case, software or firmware really means true microcode which more closely approximates logic design than anything else.) It is true that writing microcode does not have to be all that difficult if the architecture has been done properly. That "if" and "properly" are both large constraints.

One additional point needs to be made with respect to microprocessors before closing. Microprocessors in their early stages were designed by whatever methods were available and were simply required to fit on a chip. Today, that simplistic approach has been replaced by the design of microprocessors to meet certain needs. Unfortunately, one of the requirements for which microprocessors have been designed is compatibility with designs from the past. Compatibility and maintaining of standards are desirable when the effect of this is to promote growth in an industry. Under some conditions, however, it is a mechanism to perpetuate our own bad mistakes. In the future, we may see microprocessors that present a more friendly environment. To date, most of them have been based on very simplistic views of processing machines. It can be argued that since most of the microprocessors are not going to be in a multiprogramming environment, what are the needs for which some of the techniques have been developed recently? The answer can be, shortening the time to produce a working system. If the majority of the system cost is still non-recurring, except in those cases of very high production, then any reduction of development cost can affect either the price of the final product or the profit to the company. There is at least one small company that I am aware of whose existence depends upon the fact that processors are not as well designed as they might be. It is called Microforth. The system they produce is strictly software. It is an interpreter package that makes it possible to program a microprocessor in a much more friendly and reasonable environment. The resulting code is somewhat larger than had it been done by a champion assembly-code programmer. However, the time required to produce the code is significantly less and the running time is not that much worse, roughly a factor of two. Of course, we all have large armies of superb assembly-code programmers that never make mistakes and are always happy to merely make simple additions to other people's code without putting in their own ideas.

In concluding this lecture, we find the same problems in the use of microprocessors as we do in the design of any other digital system based on more conventional components. The question must be answered early in the design phase, "How many of these units are we going to build and sell?" The answer to this question will guide us in the amount of time, energy and equipment that we are going to spend during the design phase. There has been an unwritten maxim in the field of computer engineering for years which states that assembly code is more efficient than higher-level language. A corollary to this is that interpreters are fundamentally bad. One might reasonably ask the following question; 1) "What does efficiency mean?" 2) What are we attempting to optimise? 3) Is it really desirable to save 100 bytes of ROM when we must put the entire chip in anyway? 4) Where do we need to make our most efficient use of resources? 5) Is the problem to minimise chips in a final production model? 6) Is the problem one of producing a device that can be sold at a low, non-recurring engineering cost in a short time? We can always improve it later. This last option has not always been available, but today it can be since the change is primarily in how many ROMs we have plugged in.

Discussion

Professor Heath commented that good computer science solutions were required for the problems of developing microprocessor software. For example, in using a manufacturer's development system he found that he had to cope with four distinct languages.

Lecture 3 : Computers and Circuits do not a system make

What is a system? Answers are many. "A collection of components which may or may not work together satisfactorily to perform some undefinable tasks." "A complex system is one in which the benefits are imaginary, but the costs are quite real." "A system is what you build when you do not know what is needed." These cynical comments have been typical of many made in the literature. The prime factor prompting these comments can be interpreted as our escalating desires for more complex and sophisticated systems. In the case of computer operating systems, the one which almost works on system N is scrapped for a much more sophisticated and complex system that will almost work on system N+1. Tragically, this has been too often the situation in our field. Still, as systems in their various forms become more pervasive in all parts of our society, our needs rightfully escalate since the very availability of the tool is changing the way we wish to do business. The problem is not building more sophisticated systems, it is building these more sophisticated systems with the same old concepts of system organisation and design.

The emerging architecture of distributed systems is a step towards a very different kind of system organisation. In the past, we have talked about computing systems as containing a central processing unit and main memory. The disk was thought of as auxiliary memory or peripheral memory. These terms, when first used about twenty years ago, were totally accurate. Unfortunately, today, they give us a stereotype that is incommensurate with the classes of systems that we need to build. Why do we share the high-speed memory of a processor? First, it does permit a form of inter-processor communication that does not need to be fully thought through at the time the design decision is made. Second, it makes it possible for programmers to finish the design in expedient ways that get the system out the door but makes it difficult, if not impossible, to analyse. We have already discussed the problems of sharing a processor by means of multiprogramming. A concept that was valid with the programming costs significantly less than the processor. The reverse is too often true today. Communication between major elements of a distributed system is one of the real areas of fundamental research and development today. As we develop more comprehensive and truly distrubuted systems, this intercommunication requirement looms larger. Such networks can be major problems. However, they offer one of the more interesting and perhaps the only solution to building more complex and reliable systems. The trick is to cause the various major elements of the system to cooperate towards a single solution, and still at the same time maintain their isolation and independence so that errors in one element do not propagate into another. Looking at such a system from the standpoint of the quantum mechanic, we are not too concerned with the local strong interaction, but the global weak interactions can kill us.

There are newer systems being built and marketed today that only a few years ago we would have all called medium to large systems. They have been produced by a very few number of people, in some cases only one. The reason for their capabilities is simply the microprocessor. The availability of more power in smaller packages has made it feasible to build quite sophisticated systems for a low price. As a consequence, they have not been the management impediment that we often think of as management controls laid on the smaller systems. Even with these small microprocessor-based systems, however, the activity of design is still inherently labor intensive. The difference is that in the case of a microprocessor-based system, often the designer has a microprocessor aid at his disposal. These aids, which we discussed in the previous lecture, are less than fully adequate. Still, the availability of an aid to an individual designer cannot be overlooked. Time-sharing was a mechanism whereby we all hoped that this kind of power could be placed in the hands of the system designer. Unfortunately, we were in the position that to design a time-sharing system we really needed a good time-sharing system. Other design aids are starting to make their appearance today. None of them is ideal. But the lesson that can be drawn from the microprocessor-based system design is that management of a project should exert control without hampering the actual design process. Many systems have gone into the field improperly structured because it was easier to do that than to fight the necessary changes through the two or three layers of review committees.

For the remainder of this lecture, I should like to examine various levels of design aids and use as examples those that I am personally familiar with. These are not answers for everyone. It is not even clear they are answers for me. They do represent various classes of aids that are of significant utility.

The need for good simulation no longer has to be sold, but getting good simulation is not always possible. Many simulation languages are currently available and are very useful in the design activity. The problem, however, with many of these languages is that they are aimed at the professional simulator. Some of the problems faced by the system designer can be cast into the form of queues, flows, etc. Others have to be cast into complex logical interactions of both a combinatorial and sequential nature. APL over the last few years has emerged as one of the more interesting simulation environments. It is quite capable of supporting all of the logical types of simulation as well as some of the flow and queueing models. Unfortunately, there are not some facilities available in APL as there are in some languages such as SIMULA. In most cases, this has not been a drawback. The SIMULA class of simulation has been useful in very large flow models. APL, on the other hand, has been very useful for simulating everything from a simple algorithm to a chip layout to a flow in a network of processors.

The world is still looking for a good implementation language. That is not to say there are not good implementation languages already around. The problem is that they always seem to exist on a machine not available to the project. With those in which I cannot escape to machine language, I have a difficult time talking about specific addresses that are required by the architecture. With those in which escape to machine languages is possible, then control of unwanted use of assembly language is virtually impossible. There is also the problem of convincing a programmer that it is not degrading to either their intellectual capability, their moral standards, or their manhood to write in a high-level language.

We have already discussed the use of small machines to support the designer. A microprocessor-development system, available from some manufacturers, has been discussed in the last two lectures. There are uses for other small machines to aid the designer. In particular, during the development of a bit-slice-based system, machine aids can be invaluable. The same can be said of any system in which a fully checked-out processor is not one of the components. Such a small machine can be used to simulate the usual panel of lights, buttons and switches that all such checkout environments now require. The advantage of using a small machine is that all actions taken can be logged so that if erroneous results are produced, it is possible to find out the cause. There are a number of minis that can meet this requirement. A specific unit found useful in our laboratories has been the Hewlet-Packard 9825 programmable calculator. This is only a mini with a built-in interpreter and interrupt structure that can be invoked in a high-level language. Since in a particular configuration it is aimed at the process control and instrumentation environment, it is more than adequate for many of our tasks. (It has 64 parallel I/O lines, with the ability to transfer data at up to 400,000 words per second.) The advantage of using a machine of this type is that the designer is primarily concerned with solving his design problems rather than those of the manufacturer of a larger system either of hardware or software.

There are a number of instruments coming on the market that aid the designer in analysis of complex logic. Logic analysers, in-circuit emulators, logic tracers, are all examples of these forms of instruments. Some have come from the classic area of instrumentation now being applied to the digital domain. Others, the in-circuit emulators, are an attempt to give the hardware software designers better tools based on microprocessor technology, and aimed at microprocessor-based systems.

All of the above are aids that are of specific use at various points of the design process. Each attacks a particular problem. What is not currently available is any aid that permits the design process to be operated closed loop. In other words, a system in which it will be possible to determine the full effect of a design decision, and depending upon the results of the analysis iterate the design in a rational, coherent, and organised fashion. Such tools are still in the research, or at best, advanced development stage; many are based on network analysis forms. These graph-theoretic models first emerged in the early '60s. Their use is still confined almost entirely to university research. The system I am most familiar with is one called LOGOS. It was the aim of this system to make it possible for the designer to defer the hardware/software trade-offs in a system as late as possible. Further, it was aimed as a system to aid the designer in hardware trade-offs in terms of classes of equipment at both macroscopic and microscopic levels. The LOGOS class systems were all developed within the environment of a time-shared central system. The most compact version of such a system is currently running at Heriot-Watts University at Edinburgh on a large PDP-11. These systems, however, do appear to give a type of support and design environment lacking in any other systems. Trade-off analysis and flow analysis can both be conducted in the same environment to determine the effect of one upon the other. It is not, at present, easy to translate the results of such systems into hard-running code for a particular processor or specific logic design. The latter is in somewhat better shape. It might prove interesting to see what such systems could do if proper "compilers" were available to produce running code. Equally, it might be interesting to see what the compatibility would be between such a design, analysis tools, and proper implementation languages. Finally, what would the effect be in our field if design tools of this power could be microprocessor based and placed into the hands of individual designers?

Design is one of the highest aspects of engineering art and system design can be one of the highest aspects of all design. It is doubtful that there is an "ultimate" solution to the system design problem since it, in itself, is a system. We can yearn, however, for more systematic approaches to system design. We can take comfort in the fact that this has been a "cry" for centuries. We could even imagine that some lowly draughtsman in Egypt 4,000 years ago, while slaving over his stone tablet, raised the question, "Why can't we use softer stone for the rough draft?"