



Proceedings
of Caltech Conference on
Very Large Scale Integration

Computer Science Department
California Institute of Technology

3340:TR:79

PROCEEDINGS

of the Caltech Conference on
VERY LARGE SCALE INTEGRATION

held at the
California Institute of Technology

22 - 24 January 1979

Organized by the Caltech Computer Science Department
and the Caltech Industrial Associates Office

Editor: Charles L. Seitz

(c) Copyright, 1979, Caltech Computer Science Department, all rights reserved

TABLE OF CONTENTS
and Conference Program

FOREWORD

v

KEYNOTE SESSION

Are We Really Ready for VLSI?	3
-------------------------------	---

Gordon E. Moore

VLSI and Technological Innovation	15
-----------------------------------	----

Carver A. Mead

INVITED SPEAKERS SESSION

Submicron Fabrication (abstract)	30
----------------------------------	----

Edward D. Wolf

Device and Circuit Design for VLSI	31
------------------------------------	----

Amr Mohsen

The Impact of VLSI on Design and Design Automation	verbal
--	--------

<i>Roy L. Russo</i>	presentation
---------------------	--------------

Mathematical Aspects of VLSI Design	55
-------------------------------------	----

Martin Rem

Let's Design Algorithms for VLSI Systems	65
--	----

H.T. Kung

FABRICATION SESSION

91

Chairperson: Lynn A. Conway

On the Use of Nonvolatile Program Links for Restructurable VLSI	95
--	----

J.I. Raffel

A Subnanosecond LSI Family for Mainframe Technology (summary)	105
---	-----

H.H. Muller, H. Stopper, R.K. Tam

A Simple Two-Layer Aluminum Metal Process for VLSI <i>Robert J. Huber</i>	113
A Computer Study of Electron-Electron Interaction in High Density Electron Beams <i>Tateaki Sasaki</i>	125
Implementing VLSI Systems in a Research Environment <i>Robert W. Hon</i>	139
INNOVATIVE LSI DESIGNS SESSION 147	
<i>Chairperson: Robert F. Sproull</i>	
VLSI System for SAR Processing <i>Danny Cohen, Vance Tyree</i>	151
Logic-Enhanced Memories: An Overview and Some Examples of Their Application to a Radar Tracking Problem <i>W. Michael Denny, Ernest R. Buley, Earl Hatt</i>	173
WSI Distributed Logic Memories <i>R.M. Lea, M. Streetharan</i>	187
Systolic Priority Queues <i>Charles E. Leiserson</i>	199
Object Oriented Raster Displays <i>Bart Locanthi</i>	215
Storage Management in a LISP-based Microprocessor <i>Guy Lewis Steele Jr., Gerald Jay Sussman</i>	227
COMPUTER-AIDED DESIGN SESSION 243	
<i>Chairperson: William R. Heller</i>	
VLSI Design Methodology: The Problem of the 80's for Microprocessor Design <i>Bill Lattin</i>	247
Requirements for a Research-Oriented IC Design System <i>Jonathan Allen</i>	253

Hierarchical Design for VLSI: Problems and Advantages <i>W.M. vanCleempunt</i>	259
Data Base Considerations for VLSI <i>L.W. Leyking</i>	275
Bristle Blocks: A Silicon Compiler <i>Dave Johannsen</i>	303
Silicon Compilation - A Hierarchical Use of PLAs <i>Ron Ayres</i>	311
ADL: An Hierarchical Logic Design Language <i>Hilary J. Kahn, A.K. Burston, D.J. Kinniment</i>	327
 SELF-TIMED LOGIC SESSION	
<i>Chairperson: Charles E. Molnar</i>	341
Self-Timed VLSI Systems <i>Charles L. Seitz</i>	345
Characterization and Scaling of MOS Flip Flop Performance in Synchronizer Applications <i>Thomas J. Chaney, Fred U. Rosenberger</i>	357
Synchronization Strategies <i>M.J. Stucki, J.R. Cox, Jr.</i>	375
The Trimosbus <i>Ivan E. Sutherland, Charles E. Molnar, Robert F. Sproull, J. Craig Mudge</i>	395
Timing Considerations in Logic Arrays and Their Importance to Self Timed Digital Circuits (abstract) <i>Suhas S. Patil</i>	428
 ARCHITECTURE SESSION	
<i>Chairperson: J. Craig Mudge</i>	429
VLSI and High Performance Computers (abstract) <i>Herbert Schorr</i>	433

Single-Chip Computers, The New VLSI Building Blocks <i>Carlo H. Sequin</i>	435
A Cellular, Language Directed Computer Architecture <i>Gyula A. Mago</i>	447
Computations on a Tree of Processors <i>Sally A. Browning</i>	453
A Data-Driven Machine Architecture Suitable for VLSI Implementation <i>A.L. Davis</i>	479
Area-Time Complexity for VLSI <i>C.D. Thompson</i>	495
Direct VLSI Implementation of Combinatorial Algorithms <i>L.J. Guibas, H.T. Kung, C.D. Thompson</i>	509
How to Use 1000 Registers <i>Richard L. Sites</i>	527

FOREWORD

Integrated circuit fabrication technology has advanced in the past 15 years from the capability of producing a few to producing tens of thousands of switching devices interconnected on a single silicon chip. It is well understood what has made the advance from the integrated logic gate to the integrated processor possible. Improvements in the fabrication technology have reduced the density of defects, which makes larger chips feasible. At the same time these improvements have increased the density of circuitry by making transistors and wires much smaller. Reduced size also results in faster and/or lower power operation.

This progress has been so steady that the microelectronics industry has come to expect and to count on it. Very Large Scale Integration (VLSI) is a projection, a promise, a prophesy, that the evolution from Small Scale Integration (SSI) to Medium Scale Integration (MSI) to Large Scale Integration (LSI) will be followed by something even grander. Indeed, there is every reason to expect the trend to increased function on single chips to continue. The physical theory of semiconductor devices indicates that transistors as small as 1/20th of today's typical 5 micron dimensions would still function. Hence an additional increase in density of about 400 times is possible before any fundamental limits are reached. There is a practical rather than absolute limit to the size of chips, based on the best balance between yield and complexity. Optimum chip size is generally projected to increase.

This scaling of the fabrication technology promises such enormous returns in computing, defense, instrumentation, communication, and consumer electronics applications that aggressive research programs have been initiated in universities and in industry both here and abroad. One major segment of this

research is directed toward the development of fabrication and lithographic techniques to produce circuits whose parts are even smaller than a wavelength of visible light. While it is the fabrication technology which appears most closely related to achieving the goal of VLSI, even the present art may be more severely limited by the ability to design than the ability to fabricate circuits. The other major segment of VLSI research is concerned with methods for managing the design of systems composed of very large numbers of switching elements, and information system applications and architectures which are well adapted to VLSI.

This conference was organized to provide a broad view of the research efforts underway both in industry and in universities. Caltech has been a pioneer in education and research in integrated circuit and system design, and we at Caltech are pleased to have been able to provide a forum for the diverse and interesting work reported at the conference and in these *Proceedings*. The attendance of nearly 500 people, more than three times as many as had been expected in our early planning based on the largest previous attendance at a conference sponsored by the Caltech Industrial Associates, was gratifying in the size of the response, but particularly in its distribution. The mix of people from universities, government, and industry was as apparent at the conference as it is in the authorship of the papers in these *Proceedings*.

The opening session of the conference was devoted to presentations by keynote and invited speakers, and was arranged to provide an overall view of the economic, engineering, scientific, and mathematical issues and aspects of the field. Although we invited these leaders in the field to speak without any obligation to prepare written material for these *Proceedings*, most speakers have provided papers or abstracts derived from their notes or from tape transcriptions.

The next two days of the conference were devoted to five

technical sessions: fabrication, innovative LSI designs, computer-aided design, self-timed logic, and architecture. The papers presented were selected from nearly twice as many submitted. The organization of these sessions and the paper selections were the responsibilities of the session chairpersons, Lynn A. Conway, Robert F. Sproull, William R. Heller, Charles E. Molnar, and J Craig Mudge. The chairpersons have also provided an introduction to each of the sessions for these *Proceedings*.

The overall organization of the conference was undertaken by the undersigned pair, members of the Caltech computer science faculty, but most of the work was done by Tom Walters, director of the Industrial Associates Office, by the Caltech public events people, and by our secretaries, Donna Glaviano and Janice Patterson. These *Proceedings* were edited by Chuck Seitz, with a lot of help from Donna Glaviano and Chris Hankins.

Charles L. Seitz

John P. Gray

KEYNOTE SESSION

Keynote speakers:

Gordon E. Moore, President, Intel Corporation

Carver A. Mead, Professor of Computer Science, Electrical
Engineering, and Applied Physics, Caltech

Are We Really Ready for VLSI²?

Gordon E. Moore
Intel Corporation

A tremendous interest in VLSI is all around us. There is much talk of electron-beam and X-ray lithography tools to achieve VLSI's submicron structures. In all of the VLSI discussions, the implication is that it will allow us to keep on enjoying the same kind of fantastic low-cost advantages previous IC technologies have provided us in electronic products. Perhaps this may become true, but if the semiconductor industry had a million-transistor technology like VLSI, I'm not so sure it would know what to do with it. Besides products containing memory devices, it isn't clear what future electronic products that take advantage of VLSI will be.

Examples abound of products with decreases in cost from 10 to 100,000 fold, made possible by progress in semiconductor integration levels. Each increase in integration level has opened up new applications, and in several instances developed completely new industries. As semiconductor device technology evolved from discrete, to small-scale, to medium-scale, and through large-scale integration levels, product advantages have multiplied. Doesn't it seem a matter of straightforward calculation that an order-of-magnitude increase in IC device complexity should result in many of the same product advantages? Perhaps, if the products are memory related.

Memory is certainly one function that can be used in large chunks, assuming that the cost/bit will be low enough to make this possible. Single-chip microcomputers could be extended with more memory on the chip. But even here, memory modularity at some size becomes important, thus limiting the amount of memory usefully incorporated on chip.

Beyond memory, I haven't the slightest idea on how to take advantage of VLSI. In fact, the semiconductor industry is not now process-technology limited for non-memory products. How to best make use of the processing technology is really what the problem is.

Criteria for Success

Several things are required to produce a successful product, and processing technology is only one of them. (Successful product means a product that can be sold at an acceptable price to both maker and user.) Figure 1 illustrates the process of creating an LSI IC product. Each of the blocks in the figure is made up of a number of complex factors. For example, the "design" block includes the design of the process as well as that of the product. Process design requires a description of the processing sequence, the layout rules, and the electrical description of the elements of which it is composed. Product design of a complex structure requires logic and circuit designs, mask layout, and design verification. Any one of the aforementioned factors can be a formidable barrier.

At some point in the past, each of the blocks listed in Figure 1 had been a limiting factor in the success of semiconductor devices. For instance, during the first decade of the transistor, the main limitation in its successful implementation was no less than processing technology. The technology for diffusion and for making contacts had to be developed to make the transistor a reality, a device whose electrical requirements were fairly easy to define. Similarly, in the early days of the integrated circuit, processing technology was also the limiting factor to success. Features such as isolation structures had to be developed to make the IC a reality. Probably, the classic case is that of the insulated gate field effect transistor -- a device which a group at Bell Laboratories was trying to make when in the process they got hung up on something called surface states, thus leading to the invention of the point-contact and junction transistors. It

wasn't until 15 years later that the semiconductor industry learned how to manufacture a stable MOS device, and even later before it understood why.

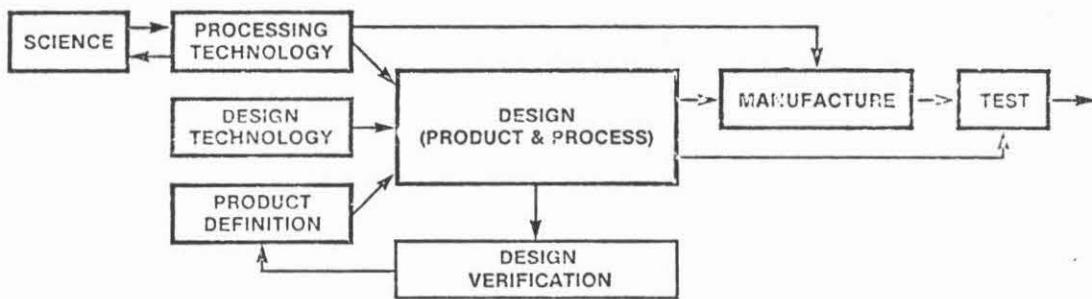


Figure 1.

Once the basic process steps were in place, progress in making ICs in ever more complex structures moved along rapidly (Figure 2), in an exponential fashion. The curve in Figure 2 is essentially the envelope of IC complexity growth. Points indicated in the figure are a sprinkling of the most complex circuit types available commercially at the time indicated. Most of the circuits introduced fall well below this curve. I expect a change in slope to occur at about the present time. From the doubling of the curve annually for the first 15 years or so, the slope drops to about one half its previous value, to a doubling once every two years. This is the rate of complexity growth than can be predicted for the future.

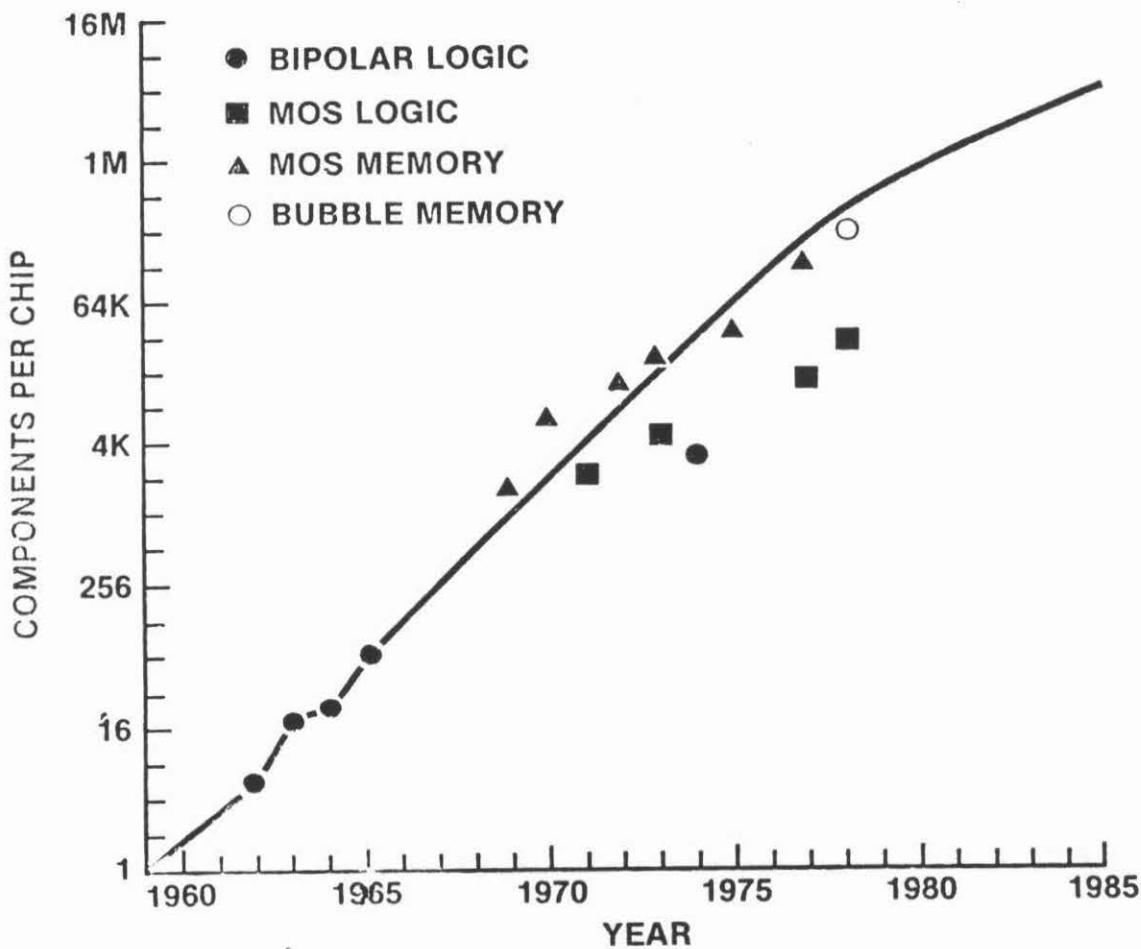


Figure 2.

The projected slowdown in IC complexity growth is caused by the semiconductor industry's loss of one of the principal factors that has allowed it in the past to increase complexities: the ability to pack more and more elements on a chip's surface by eliminating non-functional chip areas. The latest IC devices indicated on the graph of Figure 2 represent the densest ICs with the smallest amount of non-functional areas on their chips.

A Repetition of Earlier Problems

Note the gap between 1965 and 1968 in Figure 2. This gap existed because it was difficult at the time to identify any semiconductor products whose complexity came close to the limit of the time. This condition did not arise out of a lack of effort (in fact, this was a period of intense activity), but out of a problem of product definition, the very same problem the semiconductor industry is now facing as VLSI technology comes into existence. It was difficult at the time to define semiconductor products that fit the criteria for success and were near the limits of device complexity.

Two major problems faced the semiconductor industry then, as it tried to partition digital systems into complex blocks: interconnections and product uniqueness. The former problem arose from the fact that the number of leads for circuit increased so rapidly with the increase in circuit components that it went well beyond the packaging capability of that era. The latter problem resulted because the blocks tended to become unique with a resulting explosion of different part types, each required in small quantities. This condition was not conducive to successful semiconductor products.

Thus, a crisis of product definition existed. The semiconductor industry was unable to define products of high complexity that were useful in sufficiently large numbers of applications to justify their designs, and that were packagable with the available technology.

A variety of attempts to solve the problems were explored. Computer designers were asked to partition their systems into functional elements to minimize the interconnection problem. Efforts were made to confront the parts-number explosion directly. I remember at that time having discussions on how to design, manufacture and test several hundred new part types every week, in volumes of perhaps only 10 to 100 of each type. Several techniques evolved with approaches that today might be called gate arrays, wherein customized layers of metal interconnections were used on standardized diffused wafers.

The powerful computer design aids required to handle the large number of part numbers were slow in coming. Only recently have successful results been obtained. For example, IBM recently described a fantastic system utilizing direct-electron-beam writing on the silicon wafer, and a highly automatic line to handle the problem of making small quantities of a very large number of different IC designs.

In general, such efforts to solve the semiconductor industry's problems of the 1965-1968 era were not successful. The product definition crisis persisted and limited IC complexity through the mid sixties. Two things broke the crisis for the semiconductor component manufacturer, though not necessarily for the mainframe computer manufacturer; the development of the calculator and the advent of semiconductor memory devices.

The calculator was a simple system that could be partitioned into about four 40-pin IC packages, making the interconnection problem tractable. Since it was made in large quantities, sufficiently large quantities identical of components used within the calculator were manufactured to justify design costs.

As for memory, it is a universal function that can be used at the highest level of integration available. With the use of on-chip decoding, the number of leads was reduced to match available packages. What remained was for semiconductor memory to be cost competitive with established technologies for it to blossom.

Thus, the interconnection and product definition problems of the past were not necessarily solved. They were simply circumvented. The semiconductor industry developed a different set of markets in which it could keep itself busy, postponing the solution of its previous problems.

The MicroProcessor Smooths the Way

Just as the calculator and memory enabled the semiconductor industry to continue making more complex devices for certain applications, the microprocessor extended the range of use. With its general purpose architecture one could program the microprocessor to perform in a wide variety of applications providing a solution for the product definition problem.

Thus, during the 1970s, the semiconductor industry kept developing more complex memory chips to track the complexity curve in Figure 2, with microprocessor products following closely behind. Large-computer manufacturers were left to solve their own problems of part number proliferation and low-volume uses, often through the use of components with lower levels of integration. Thus modern LSI technology has not eliminated predecessor technologies of small-scale and medium scale integration. For example, the number of bipolar semiconductor devices produced continues to grow rapidly, from about 850 million circuits in 1972, to about 1.5 billion in 1974, down to a little over 1 billion during 1975-1976, and up again to about 2.5 billion last year, worldwide. The availability of high levels of device complexity has not resulted in the complete replacement of less-complex devices. Co-existence is more often the case. Even a company devoted to making LSI IC products finds that it cannot use the capability for complexity in all its products. The complexity of products introduced by the Intel Corporation, for example, over the last two years is shown in Figure 3, and can be compared to the limits of Figure 2.

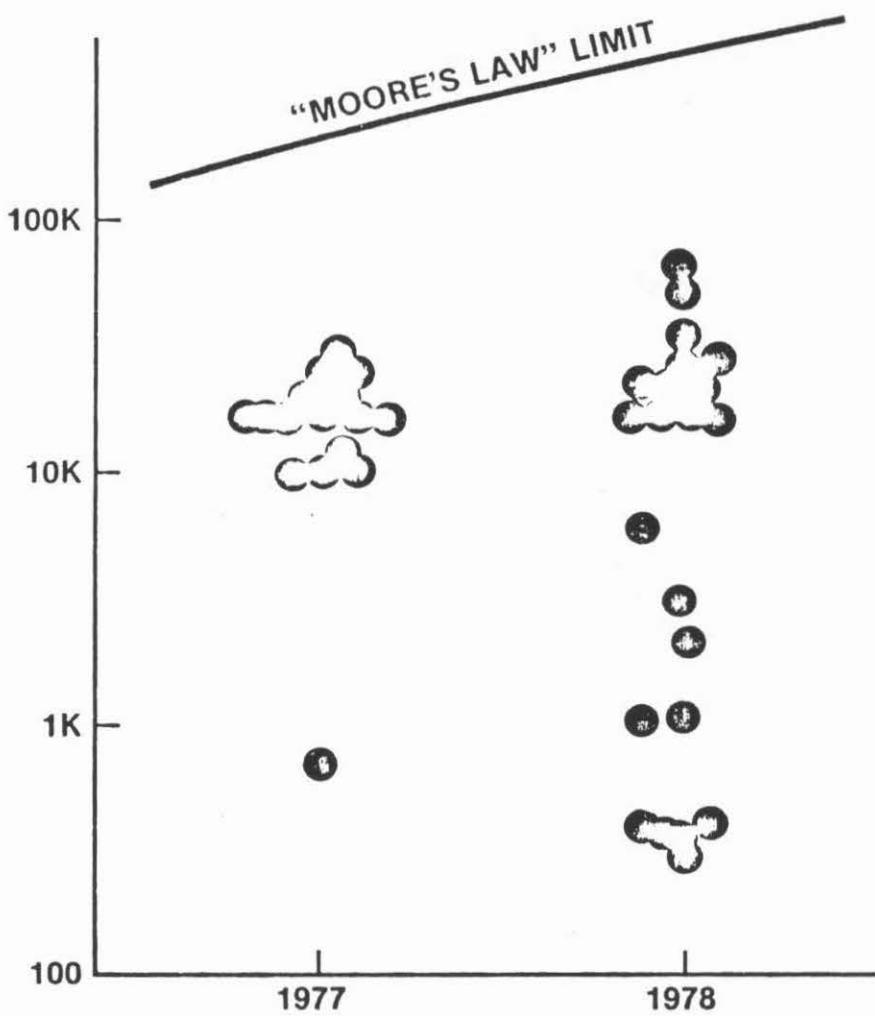


Figure 3: Complexity of Intel's Semiconductor Product
Introductions for 1977 and 1978

Note that few of the products depicted in Figure 3 are close to the "Moore's Law" limit of the same figure, many of which miss it by large factors. The most complex circuits tend to be memories, with simpler ones being microcomputer peripherals.

In Figure 3, microprocessor and complex peripheral devices tend to group around the same level of complexity. This is the level that the semiconductor industry can presently define for useful products. Although similar devices two to three times

more complex can be made, a definition of the products they would constitute is needed first. Thus we come full circle to our dilemma: how to best make use of our capability for ever more complex devices such as VLSI ICs, by properly defining such products.

Another Perspective

The product definition problem can be shown from a different perspective, by looking at the amount of effort required for product definition, design, and layout (in person-months), starting with the first planar transistor of 1959 and projecting into the future (Figure 4). This design effort is plotted on a logarithmic graph

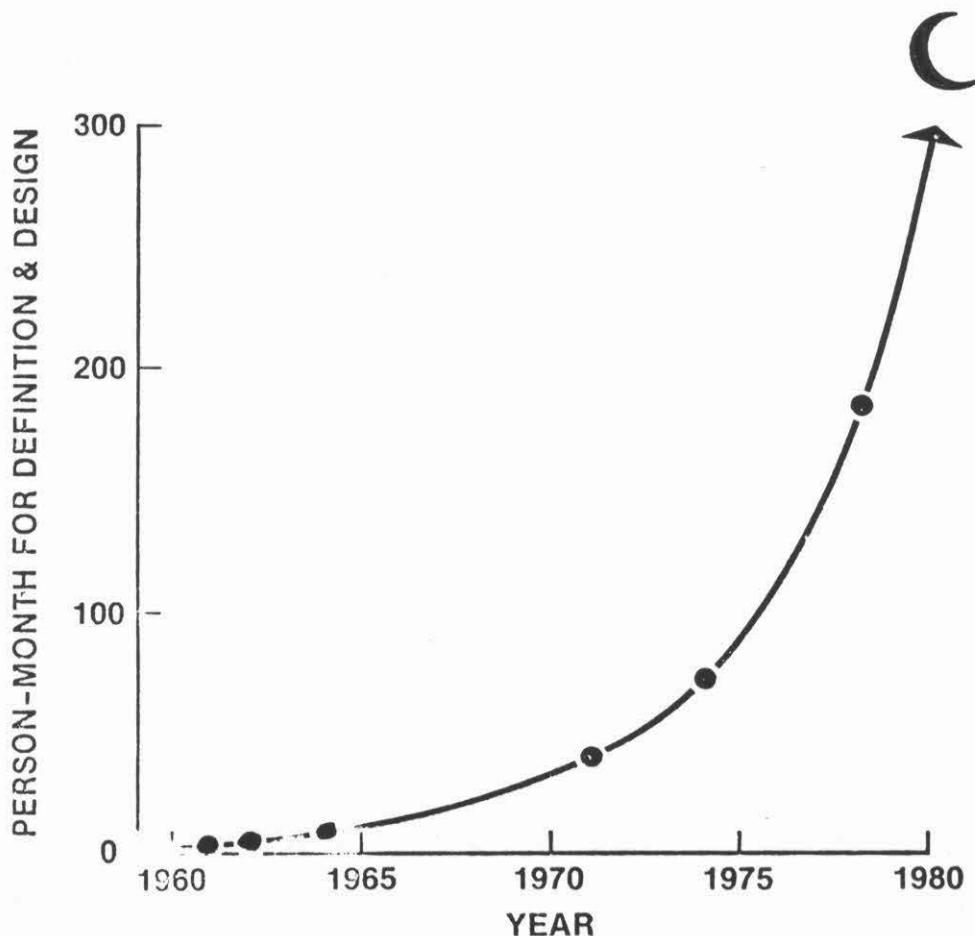


Figure 4.

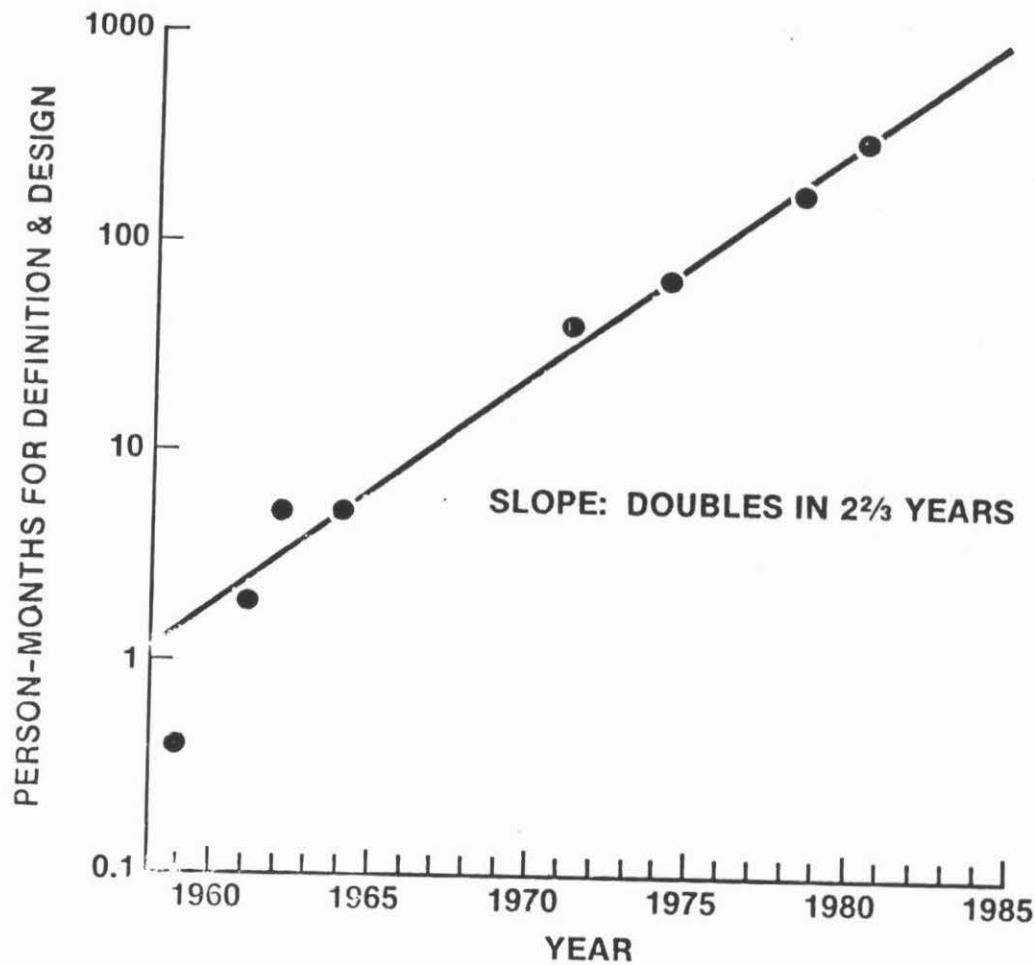


Figure 5.

in Figure 5. As can be seen from Figures 4 and 5, its growth is exponential, doubling every 2 and 2/3 years.

If it is assumed that the cost per person-month is inflating at 10 percent per year (a conservative figure considering the need for increased computer support, etc.), then the costs double every two years. We should keep in mind that device complexity is also doubling every two years, resulting in a constant cost per element to define, design, and layout complex ICs.

This cost can be contrasted with the manufacturing costs that are approximately independent of device complexity. Whereas once manufacturing costs were dominant and exceeded those of design, the situation is now reversing, with design costs becoming dominant. The implication is clear: product definition and design technology are where work is really needed. And the kinds of answer the semiconductor companies will come up with in response to these challenges will depend on the nature of their businesses.

The component supplier must have large markets across which he can amortize his high design costs. This requires high-level standardization, either at the processor level or at the very large system level. This will limit the breadth of VLSI's impact as shown in Figure 6. Only memory devices may utilize maximum complexity. Discrete devices, MSI and LSI logic functions, and LSI will remain important in future systems.

The principal capability for defining and designing LSI and VLSI products is in the hands of the systems suppliers. If product definition and design will become the important factor of the future and I believe that it will, then the systems companies may have the advantage in VLSI's success. They also have the desire to preserve existing structures such as large cumulative software investments.

The result is that a structural change is occurring in the semiconductor industry. On the one hand, component suppliers, as always, are pushing for standard products that are useful in large numbers across a broad spectrum of applications. On the other hand, an increasing number of systems companies or captive suppliers are becoming more skilled in the technology of making complex ICs. Such companies are expanding their in-house processing capabilities and are using them successfully. A few years ago, I maintained that there were only two successful captive suppliers in the world. Today, there are clearly many more.

According to the most recent compilation from Dataquest Corporation, the number of worldwide component suppliers in the semiconductor industry between 1975 and 1979 dropped by about

10 percent. On the other hand, system companies with in-house captive suppliers -- not simply R & D laboratories, but companies making products for use in their own equipment -- grew from 19 to 43 during the same period of time. Clearly, the industry is changing.

As for my original question, whether or not the semiconductor industry is ready for VLSI, the conclusion is that for maximum advantage, both suppliers of components and systems must address the problems of product definition and design. In fact, unless we address and solve these problems, as we look back on the VLSI era, we may only be able to say, "Thanks for the memories."

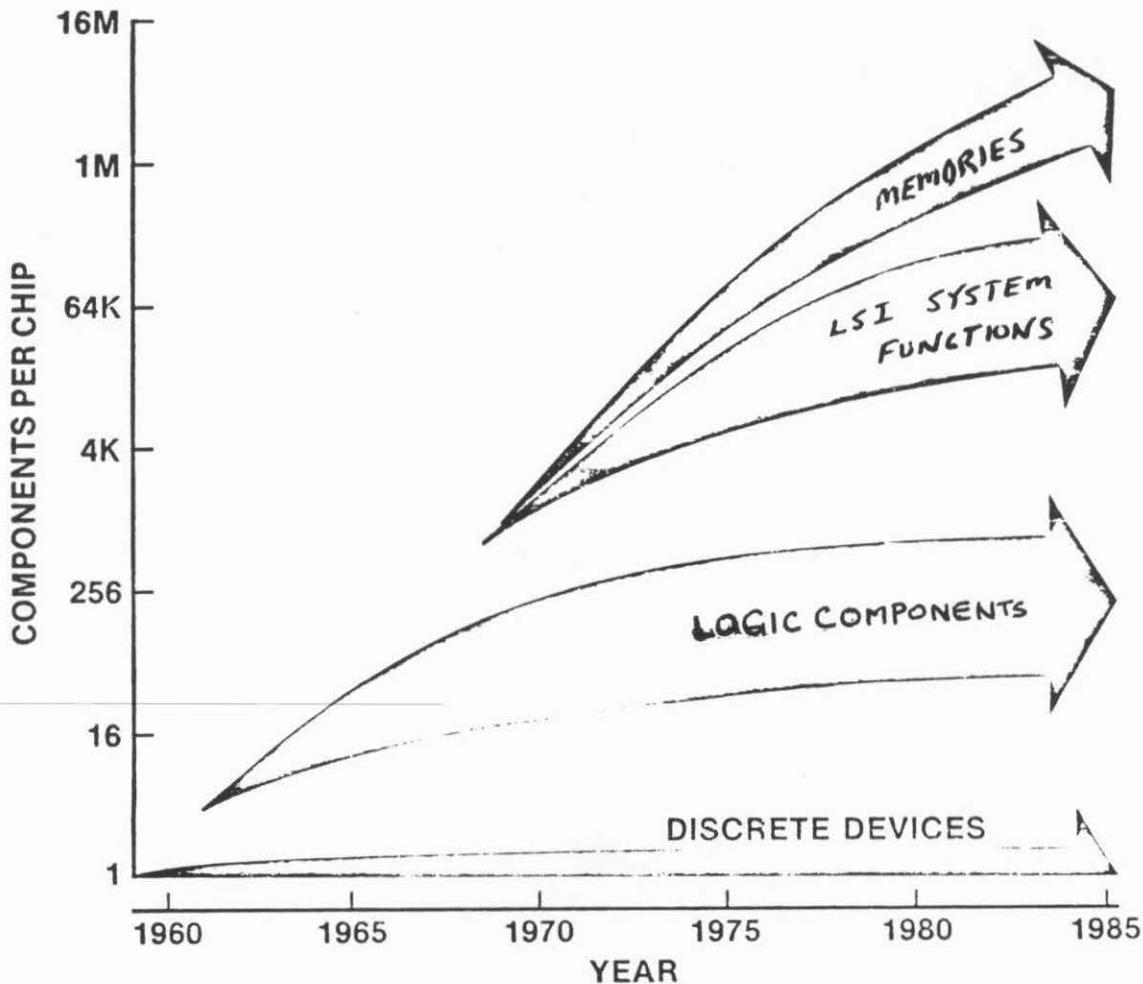


Figure 6.

VLSI AND TECHNOLOGICAL INNOVATION

Carver A. Mead

Professor of Computer Science,
Electrical Engineering and
Applied Physics

California Institute of Technology

VLSI relies on a range of disciplines for its successful implementation. Two of the most important of these are still in their infant stages.

A. Design methodologies to manage complexity.

B. Architecture of ultra concurrent machines.

Innovation in infant disciplines occurs most rapidly and successfully when a large number of small groups proceed independently under the motivation of market opportunity. In a few years, a substantial fraction of the engineering work force will have a working knowledge of LSI design. At the same time, fabrication areas are becoming more and more capital intensive. What is needed is a clean, standard interface between a multitude of small diverse VLSI design groups and a few state-of-the-art fabrication suppliers. A proposal for such an interface is presented in this article.

[Note: This article elaborates on only one of the several topics in Prof. Mead's talk at the conference. --ed]

The electronics and computer industry of the future will look radically different than it does today. Using the past as a guide, we can guess with reasonable certainty the course of future evolution. Figure 1 shows the evolution of the various components of today's silicon manufacturing business. At the bottom are the discrete transistors, diodes, rectifiers, etc. They still form a substantial fraction of the entire semiconductor business. Above them lies the small scale, medium scale and large scale standard parts integrated circuit business. This business, dealing in large volumes of standard catalogue items, will always exist and in fact will grow in the future. Riding above it, however, is a rapidly increasing segment dominated by VLSI designed by those who will take it to the end user market. This is the true world of VLSI. It will not compete directly with the other branches of the semiconductor industry just as memory manufacturers do not compete with rectifier manufacturers.

VLSI is a statement about system complexity, not about transistor size or circuit performance. VLSI defines a technology capable of creating systems so complicated that coping with the raw complexity overwhelms all other difficulties. From this definition, we can see that the way in which the industry responds to VLSI must, in fact, be different from the way it has historically evolved through its other phases.

The complexity scale implied by the new technology can be appreciated from the analogy presented in Figure 2 (1). At several points in the evolution of the technology, a typical chip has been scaled up to make the spacing between conductors equal to one city block. The circuit can then be thought of as a multi-level road network. In the mid 1960's, the complexity of a chip was comparable to that of the street network of a small town. Most people can navigate such a network by memory without difficulty. Today's microprocessor is comparable to the entire Los Angeles basin. By the time a 1μ technology is solidly in place, a chip design will be comparable to planning a street network covering all of California and Nevada at urban densities. The ultimate $\frac{1}{4}\mu$ technology will be capable of producing chips whose complexity rivals an urban network covering the entire North American Continent.

Designers are just now beginning to face complexity as a central and dominant issue of the next stage of evolution. They have not yet begun to face the capabilities that such a technological revolution brings to us. The evolution of the component fields which make up the present VLSI discipline are shown schematically in Figure 3. What is plotted here is the number of new ideas, weighted by their importance, as a function of time. Each component discipline undergoes a period of exponential growth when each new idea spawns several others. Later, a period of linear growth ensues while the interstices between the fundamental ideas are being filled in. Later, a logarithmic law ensues in which ideas are being ground finer and finer but very little conceptually new content is added. By now, the number of dramatically new ideas being added to the device physics area is small. Fabrication technology has essentially all the fundamental knowledge that will be required. Circuit and logic design have some cleverness left but that too will soon saturate. The large system design methodology is still

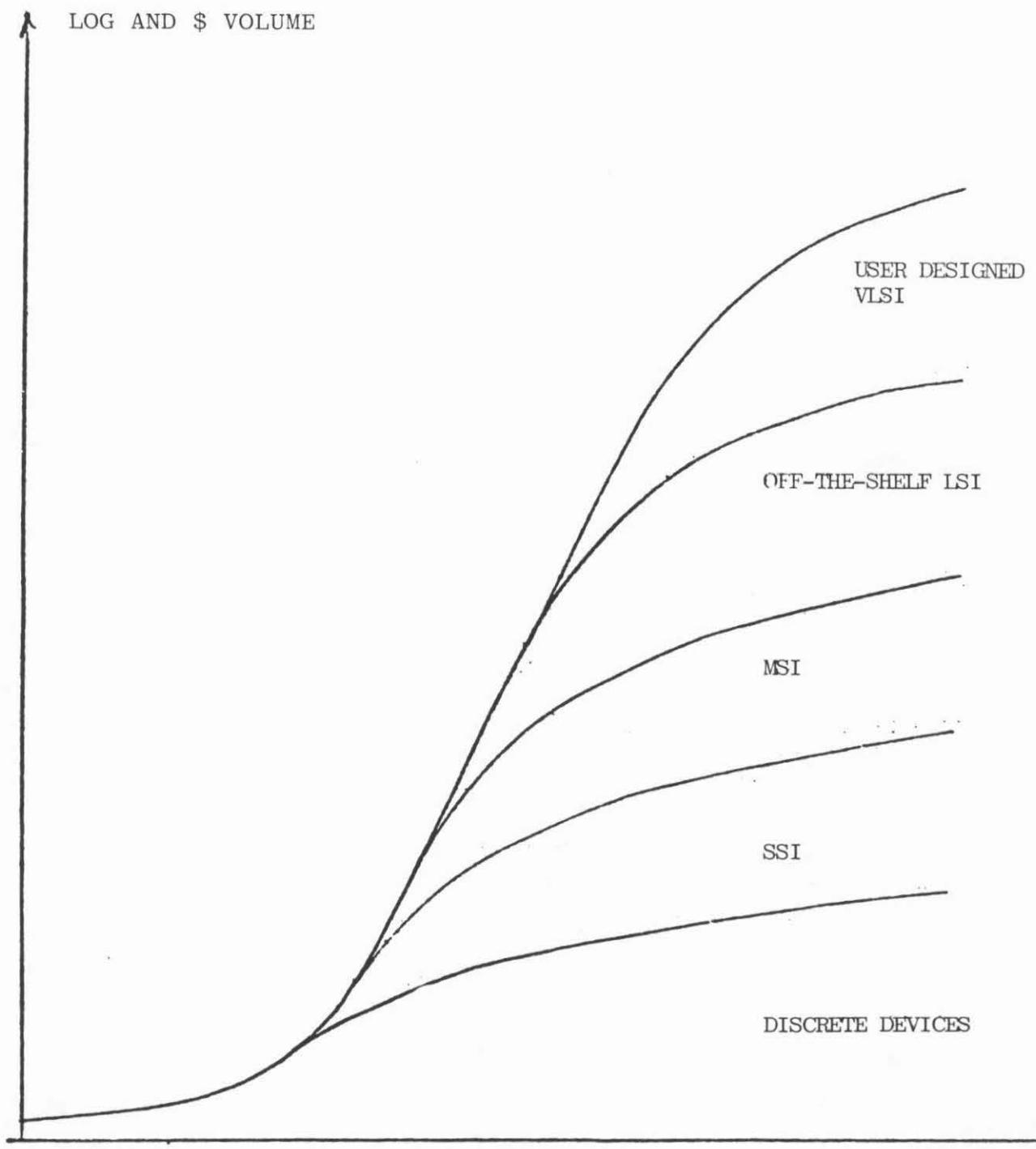


Figure 1

≈ 1963

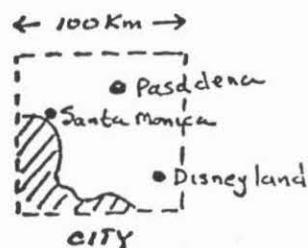
$$2\lambda = 25\mu \quad \underline{4m} \times \quad \Rightarrow \quad \square \quad \xrightarrow{\text{1 mm}}$$

$$4\lambda = 50\mu$$

 ≈ 1978

$$2\lambda = 5\mu \quad \underline{20m} \times \quad \Rightarrow \quad \square \quad \xrightarrow{5mm}$$

$$4\lambda = 10\mu$$

 $\approx 1985?$

$$2\lambda = 1\mu \quad \underline{100m} \times \quad \Rightarrow \quad \square \quad \xrightarrow{10mm}$$

$$4\lambda = 2\mu$$

 $\approx 19??$

$$2\lambda = 0.25\mu \quad \underline{400m} \times \quad \Rightarrow \quad \square \quad \xrightarrow{20mm}$$

$$4\lambda = 0.5\mu$$

\uparrow
scale factor to make
blocks 200 m apart ($5/\text{km}$ or $8/\text{mile}$)

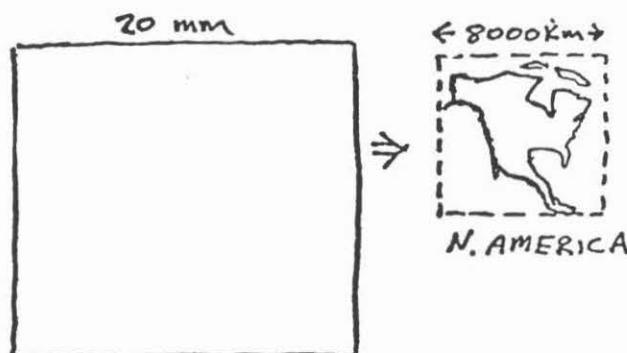
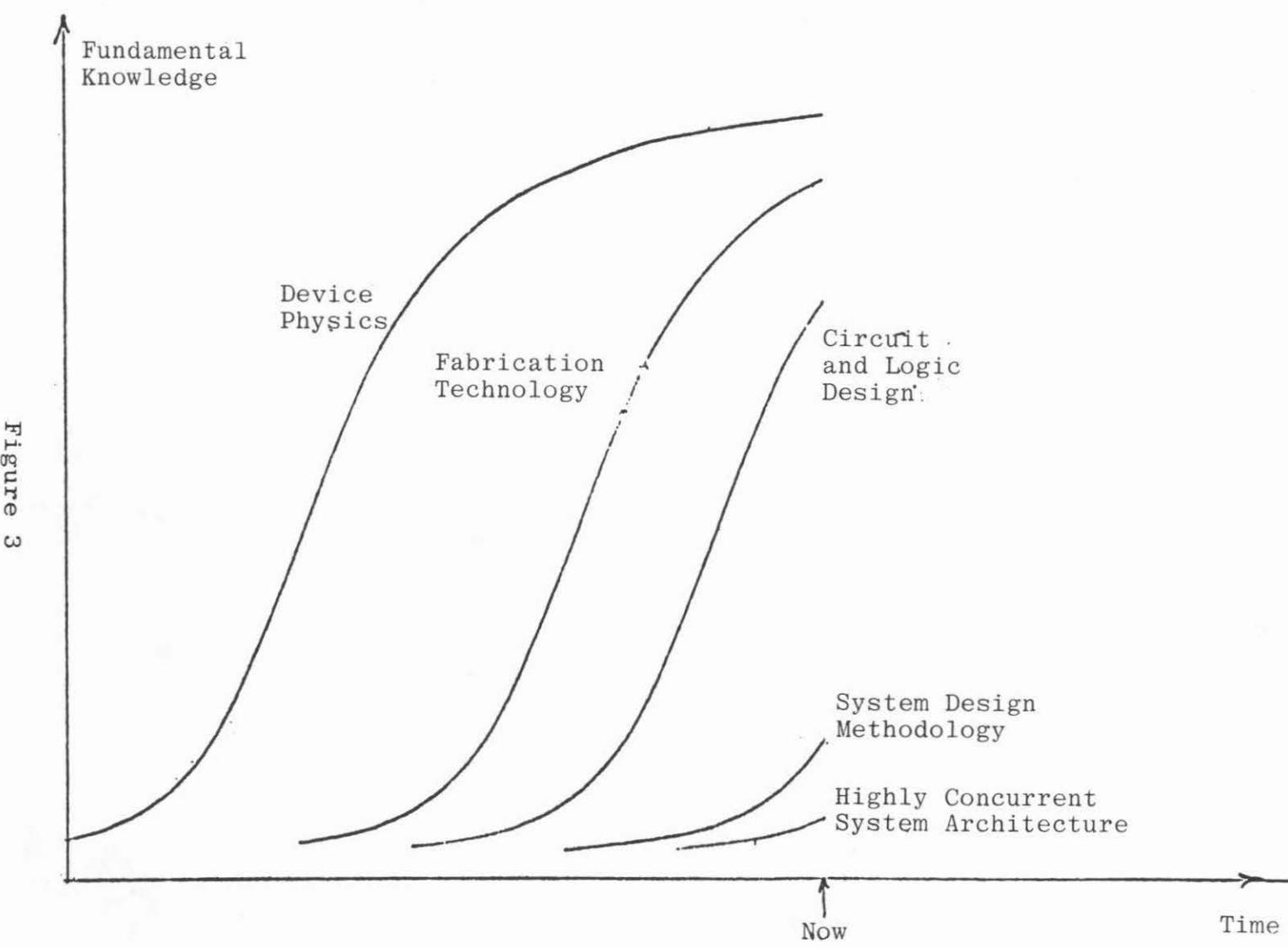


Figure 2



in its exponential phase. Many fundamental ideas have yet to be discovered. The architecture and algorithms for highly concurrent systems is even less well developed. Only a few results are known and much of the fundamental conceptual apparatus needs to be discovered. A period of very rapid growth lies ahead of us in both of these disciplines. They are central to the difference between VLSI and the current way semiconductor devices are designed.

The range of knowledge required to design integrated circuits has expanded greatly as their complexity has increased. This requirement has dramatically changed the relationship between the manufacturing technology and the design process. In the earliest times, getting the device physics right was most of the problem. The physics and fabrication technology were intimately intertwined. One person could oversee the design, the manufacturing process, and the testing as well. Later, circuit design became as important as the device design, but still one individual could work between the two disciplines. In many linear circuits today, a different process is used for each product, a heritage from earlier times. However, there has been a steady trend toward standard processes. The main early driving force was the evolution of families of logic elements such as TTL, ECL, CMOS, etc. Here, the designer could implement a number of logic functions with the same process. The trend toward standard processes and a simplified interface between that process and the designer has had many beneficial results for those who have adopted it. The design process is greatly simplified. Fabrication area logistics are greatly simplified if many products can be run using the same basic process. The maintenance of any given process is a complex and tedious job. Fewer processes result in smaller maintenance problems.

As the complexity of systems increases, the potential gain in achieving optimal designs at the system level greatly outweighs advantages to be achieved by customizing a process to a particular product. Even with today's LSI technology, a factor of a thousand to ten thousand is available if ways can be found to achieve large scale concurrency for system functions. By contrast, optimizing the process to a particular part or the design of a particular part to a specialized process may achieve a factor of two. Much of our experience with the development of software is directly applicable here, since both disciplines are fundamentally concerned with management of very large, very complex systems. A hard lesson has been learned in that arena; get the design correct at the highest level and don't yield to the temptation to suboptimize. There is nothing more useless than a very fast system which does not work.

Innovation

The semiconductor technology is composed of a set of disciplines which must be considered separately. In any given discipline, innovation proceeds along an S shaped curve such as that shown in Figure 4. In the early phases, marked (A) in the figure, progress is limited by the lack of fundamental ideas. A single good idea can make possible several other good ideas and hence the innovation rate is exponential. During this period, a single

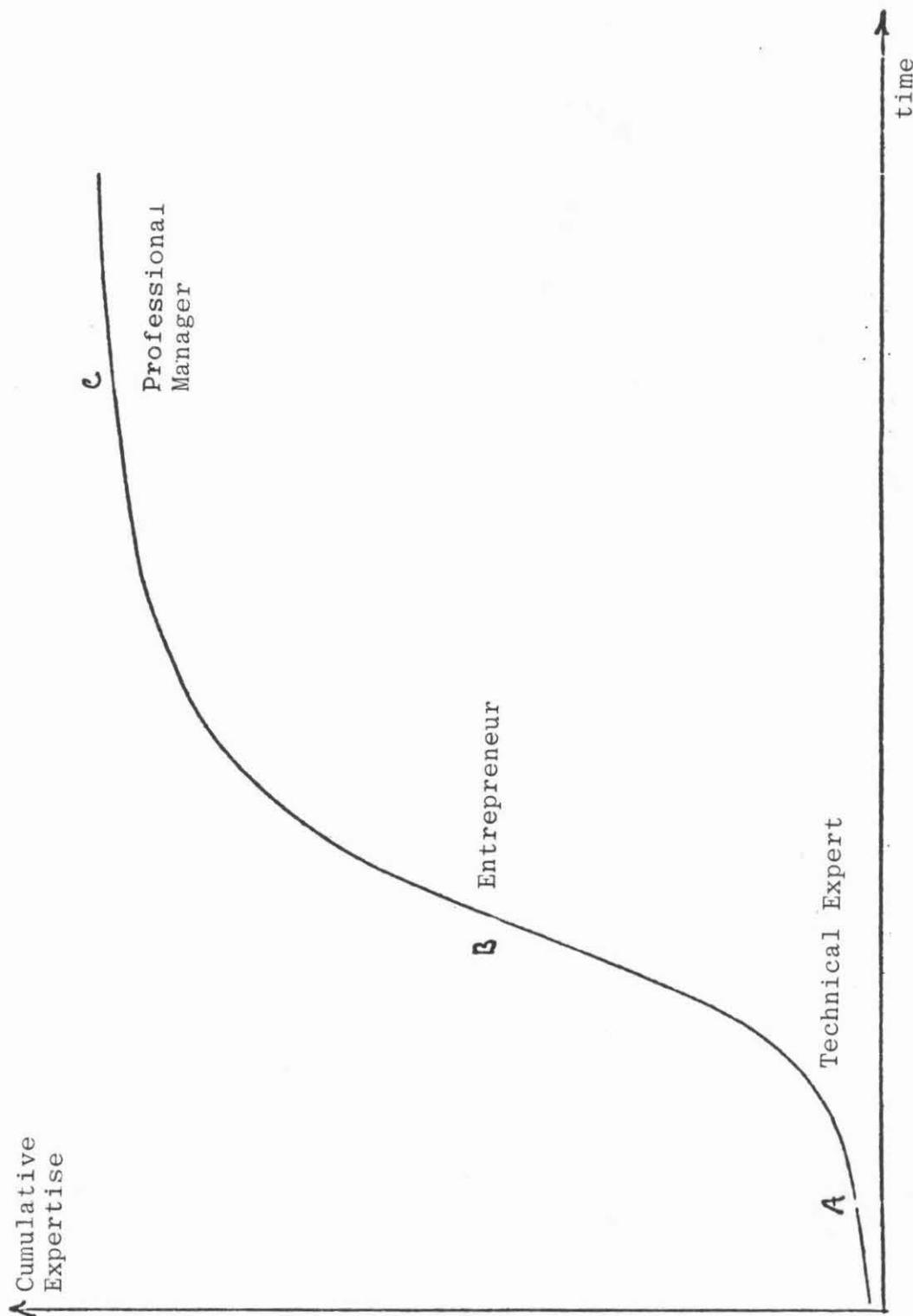


Figure 4

individual or small group of individuals can develop a viewpoint and contribute several crucial insights that set a field in an entirely new direction. It is the time during which progress is dependent upon a few visionaries within the field. During the central and most visible portion of the evolution, marked (B) on Figure 4, a linear region ensues. Here, the fundamental ideas are in place and innovation concerns itself with filling in the interstices between these ideas. Commercial exploitation abounds during this period. Specific designs, market application, manufacturing methods grow rapidly.

The field has not yet settled down at this point. Entrepreneurs backed by venture capital firms can have a large impact and achieve a dominant market share during this period. During the later stages of the evolution curve, marked (C) in Figure 4, progress becomes logarithmic in time. Manufacturing methods are refined ever further. More and more capital is expended to reduce the price of manufacturing. Here the business becomes capital intensive. Production know-how and financial expertise are the required credentials. Professional managers and large firms dominate the business.

Innovation proceeds most effectively in a large number of small groups. The problem faced by the semiconductor industry is apparent. Fabrication technology has reached its capital intensive phase. Design is still very early in its exponential phase. Historically, innovation in the industry has been spearheaded by small start-up firms and later taken up by large existing organizations. It is significant that the major suppliers of vacuum tubes did not become the major suppliers of transistors. The major suppliers of discrete transistors did not give us semiconductor memories. More recently, companies dominant in the semiconductor memory business did not bring us the multiplexed address random access memory. The microprocessor did not come from mainframe or minicomputer firms. Each of these innovations was brought to market fruition by a small start-up firm which rapidly gained market share by virtue of its innovation. Existential dominant firms were then forced to retrofit these ideas into their own product lines.

Each small group can no longer afford its own fabrication area. A start-up firm with a capital budget of one or two million dollars for a fabrication area was within the means of traditional venture capital sources. However, the same is not true for capital budgets of several tens of millions of dollars required for state-of-the-art fabrication lines in the near future. If innovation by a myriad of small groups and individuals is to carry us into the VLSI revolution, we must not expect these groups and individuals to provide their own fabrication facilities. The level of innovation required can be achieved only if fabrication is provided as a service by a few well capitalized firms.

Every time a qualitatively new element has been introduced into the industry, new business opportunities have been created. Small firms have obtained significant market shares in businesses previously dominated by large firms. The VLSI revolution we are facing is no exception. I fully expect a very large number of small firms, or small groups within larger firms, to create entirely new machine organizations and entirely new design methodologies. These will allow small, able groups to succeed in the varied market place

for systems in spite of historic dominance by capital intensive computer and semiconductor houses.

Evolution of a Product

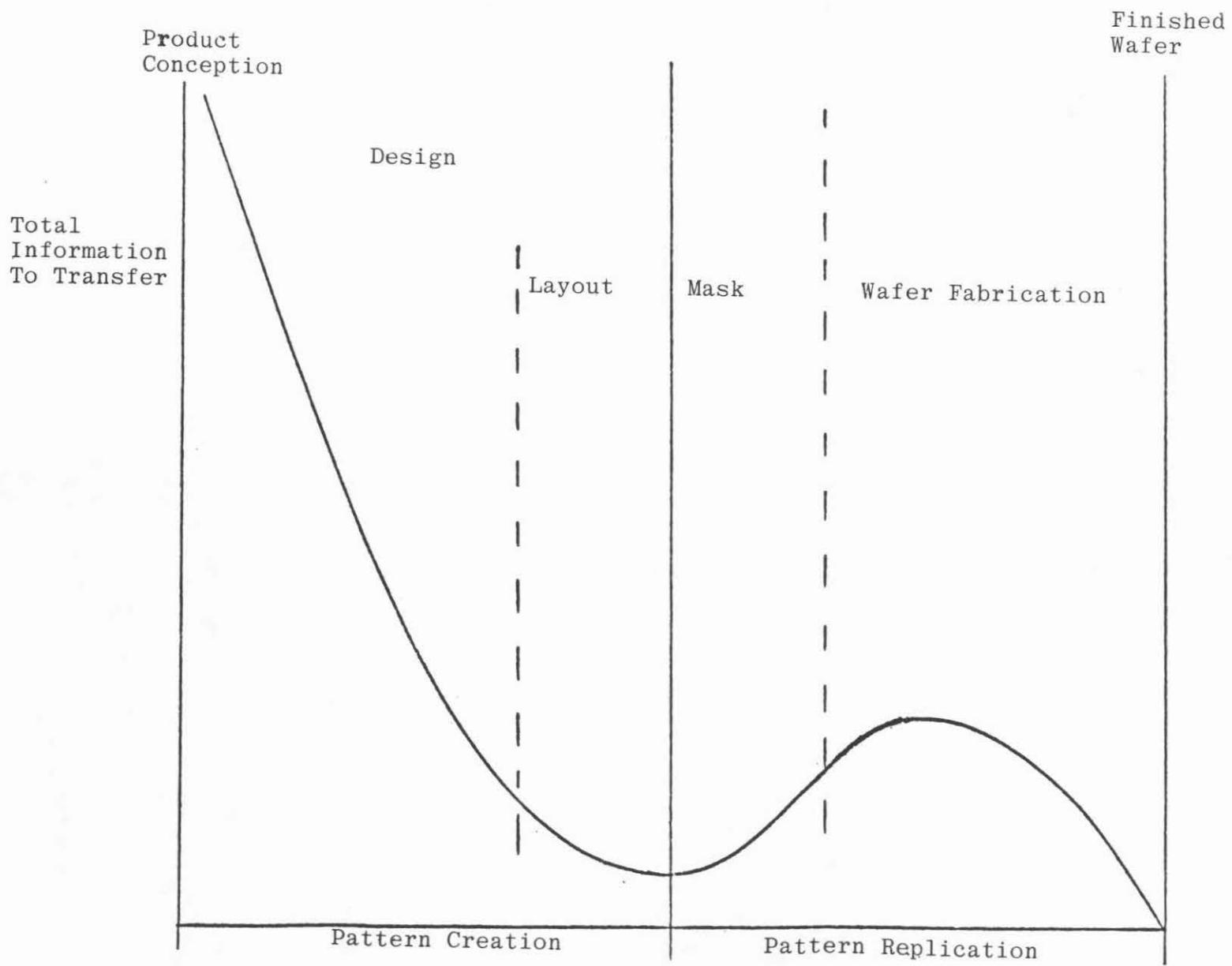
A product which is implemented as one or more VLSI chips passes through five major evolutionary phases. These are conceptual, design, layout, pattern generation and wafer fabrication.

At some stages of the evolution, it is much more difficult to transfer the knowledge required for further progress than at others. Figure 5 shows a qualitative measure of the information required to transfer the product design from one group of people to another at any given point in the design or production process. At the left hand edge of the curve, the product conception could only be transferred to someone along with essentially complete knowledge of the customer base, the economics of the business, the skills of the personnel in a particular company, cost constraints, timing constraints, etc. About half way through the design process, a block diagram could be transferred which might seem to contain only a small amount of information. However, along with the block diagram would be an enormous amount of context having to do with the myriad of special requirements not completely specified in the written specification. These performance and partitioning constraints are normally solved in an interactive manner during implementation. Examples are testing protocols used in systems developed by the same company, the way in which manufacturing constraints affect overall system design, etc. Once again, a very large amount of information accompanies a design transfer at this stage. Most of today's "custom" LSI designs are transferred to a semiconductor house at this awkward point.

There is an optimal point for the transfer of product. By the time a complete layout has been generated for the chip or chip set implementing a system function, the only information which needs to be transferred from one group to another is the patterns which represent the various layers. This point represents a true minimum in the total information transferred. For example, if one were to transfer a mask instead of data representing the pattern layers, not only would the patterns themselves need to be transferred, but also information which depends upon the details of wafer fabrication process; whether the process uses positive or negative photo resist, how much the lines or spaces of the various layers should be shrunk or expanded to compensate for aberrations introduced in a particular process, etc. None of these details need to be transferred if a data file representing the basic patterns is transferred from the designer to the factory. As the wafers proceed through the fabrication process, each layer is lithographed into the silicon and the amount of information needed for the next processing step decreases. Finally, when the wafers are finished, they may be returned to the designer as fully instantiated artifacts without any additional information.

The minimum in information required to transfer between layout and pattern generation is no accident. This is a very special point in the evolution of a product. It is the end of the design process and the beginning of a

Figure 5



pattern replication process. Everything to the left of that point has been involved with the specifics of a given product. Every action to the right of that line does not depend upon the specific product, but only upon the process by which the product will be replicated. It is thus the seam between product creation and product replication. To use a familiar analogy in the motion picture industry, those to the left hand side of the line are the producers, the stars, script writers and photographers. Those on the right hand side of the line are the film manufacturers and film processing laboratories. By analogy, we are led to ask what corresponds to the ASA number and color temperature specifications which are used to interface the two worlds of photography. Life would indeed be simple if such a clean interface could be formed between those creating designs and those printing them on wafers of silicon.

An Interface Proposal

As one might expect, the world of silicon is indeed more complicated than the world of film. However, not by as large a degree as the popular image would cause one to imagine. It is possible, with well developed standard processes, to establish a standard interface to almost all fabrication areas running that process. Such an interface requires a remarkably small amount of information to be passed across the boundary.

At Caltech, we have, over the past ten years, been working in collaboration with industry, and more recently with other universities, to develop such a clean interface to wafer fabrication. In the process of implementing 30 or so chip designs, we have interacted with ten different fabrication areas and six mask shops. Although the early interactions were very ad hoc in nature, there has recently emerged a clear vision of how such an interface can be made to work. We are convinced that a modicum of effort expended by those operating fabrication areas can drastically reduce the amount of effort required for user groups to transform designs into silicon. What is required for such an ideal interface to a standard wafer fabrication process? Such an interface consists of three specific, well defined objects.

1. Geometric Design Rules.
2. A Standard Data Format.
3. A Standard Test Chip.

A set of geometric design rules for nMOS silicon gate technology which allows designs to be run on any one of a large number of commercial fabrication areas is given in Figure 6 from reference 2. These rules have been defined in terms of a minimum length unit lambda, which can be selected to conform to the smallest dimensional tolerance in the process. It is thus scalable in such a way that it can follow the dimensional evolution of the process with time. In this way, changes in the geometric resolution of underlying fabrication steps can be taken advantage of without changing the chip design. The design rules used at Caltech and other universities have not changed in form in the last ten years. Over that period of time, the length unit lambda has changed more than a factor or two.

A standard data format is needed to transfer design files to a given fabrication area which uses a given pattern generator. Most output data formats are biased toward a particular output device. A university and industry group has recently developed an intermediate output language which is not biased to any particular output device or design system. It is known as the Caltech Intermediate Form (CIF) and is now used by a number of participating universities and industrial research organizations. The detailed description by Sproull and Lyon is given in reference 2.

The third component of a standard process interface is a standard test chip. This chip can be automatically inserted by the mask/fabrication supplier into the array of product chips at a number of places on every wafer. It contains patterns for process control and characterization of yield, reliability, circuit performance, and system performance. It must be the sole subject of the contract between a fabrication line and its users. In this way, the fabrication people are not blamed for design failures and vice versa. Recent excellent work at the National Bureau of Standards (3) and the Department of Defense (4) has brought this goal within reach. For each standard process, a standard test chip can be made available to all participating fabrication areas.

All semiconductor manufacturing organizations internally operate with the three pieces of interface lore discussed above. However, these objects are not common across corporate boundaries. Although those in many corporations are very similar, they are viewed as highly proprietary and are closely guarded secrets. Having such lore in the public domain is a key factor in assuming rapid innovation in the design of VLSI systems.

The use of a standard interface to a standard process has both costs and benefits. There is no doubt that standard processes do exist which are widely accepted within the industry. There is also no question that a standard interface to such processes could be achieved that would greatly simplify the interaction of designers, producers of design equipment, producers of pattern generation equipment, and managers of fabrication areas. Initially, the use of a standard process and common set of design rules results in lower density and speed than that possible when the product design and process are mutually optimized. Three factors minimize the penalty from this source. 1. Since the rules can be scaled, the product can be debugged with a process available immediately. By the time a product reaches the market, it can take advantage of a high density and higher performance. 2. Fabrication engineers are not distracted by a number of slightly different processes, and can concentrate on the evolution of a single, most advanced, standard process. 3. Design time can be dramatically decreased, getting product to market earlier. Product market life is larger, thus amortizing design cost over a larger number of units.

The central advantage of a clean interface is to allow designers to optimize design methodology and algorithms and architecture while fabrication engineers independently optimize fabrication processors. Such an interface has not been seen as important in the industry until recently because design

was not a major stumbling block in the road to implementing integrated circuits. The rapidly increasing cost of the design of complex systems, together with the enormous potential payoff to be achieved by the use of large scale concurrency in achieving system functions, means a major revolution in the semiconductor industry.

Opportunities exist for both operators and users of silicon fabrication facilities. Service facilities can be very profitable, since their costs are highly predictable and the market base is very broad. These facilities will become very much like the raw silicon wafer suppliers of today. High volume, high profit and low risk. Those firms engaged in the system design business will be completely different. They will be small, and must live by their wits in a constantly changing, enormously competitive industry. Many will be called, but few will be chosen. Those that in fact succeed will form the new cutting edge of an entirely reborn industry. It is an exciting future, but our ability as a nation to undertake such an adventure is dependent upon our willingness to create an available, state-of-the-art fabrication service, available to all in the field who need it.

REFERENCES

1. Seitz, Charles; "Self-Timed VLSI Systems"; from the Proceedings of Caltech Conference on VLSI, January, 1979.
2. Mead and Conway; Introduction to VLSI; to be printed by Addison-Wesley, limited printing by authors, 1978.
3. Buehler, M.G., Grant, S.D. and Thurber, W.R.; Journal of Electrochemical Society, vol. 125; p. 650; 1978.
4. Jurdenoek, R.T., Baire, Jr., H.F., Fromen, G.J.; IEEE Transactions on Elec. Dev. ED 25; p. 873, 1978.

INVITED SPEAKERS SESSION

SUBMICRON FABRICATION

E. D. Wolf
National Research and Resource Facility
for Submicron Structures
Cornell University, Ithaca, NY 14853

ABSTRACT

Submicron fabrication is the broad interdisciplinary activity that produces microstructures with feature sizes less than one micrometer. Much has been published on the fabrication of custom integrated circuits using advance lithography and pattern transfer techniques. This talk reviewed some of the limitations and advantages of submicron fabrication. Also a brief summary of the objectives, equipment status and accomplishments of the new National Submicron Facility at Cornell University were discussed.

DEVICE AND CIRCUIT DESIGN
FOR VLSI

by

Amr Mohsen*

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

ABSTRACT

A review of the device and circuit design complexity and limitations for VLSI is presented. VLSI device performance will be limited by second order device effects, interconnection line delay and current density and chip power dissipation. The complexity of VLSI circuit design will require hierachial structured design methodology with special consideration of testability and more emphasis on redundancy. New organizations of logic function architectures and smart memories will evolve to take advantage of the topological properties of the VLSI silicon technology.

* The author is also on the faculty of California Institute of Technology

I. INTRODUCTION:

As semiconductor technology has evolved from discrete to small-scale to medium-scale and through large-scale integration levels a rapid decrease in the cost per function provided by the technology have opened up new applications and industries. Today there is a large interest in the next phase of integration: very large scale integration (VLSI). The semiconductor technology will be able to fabricate chips with more than 100K devices ⁽¹⁾ in the 1980's. The continuous scaling of the semiconductor devices and increase in components counts on a single chip is resulting in more complex technology developments, device and circuit designs and product definition. In this review paper, projections of how device technology will evolve in the future and the problems and limitations of device and circuit design for VLSI are presented.

VLSI TECHNOLOGIES:

In Fig. 1 a summary of the performance of the major technologies available today for LSI's is illustrated (2,3). For bipolar devices two families: the Emitter-Coupled logic (ECL) and Integrated Injection logic (T^2L) are plotted. For MOS devices the NMOS, CMOS and SOS technologies are shown. The performance of GaAs MESFET and silicon MOSFET logic are also plotted for comparison although those devices are not presently used for LSI. The speed of the bipolar and MOS logic families are limited by the transit time of the carriers across the active region of the devices (the base for the bipolar transistor and the gate for the MOS transistor), the gate configuration of the logic family (for example enhancement load, depletion load or CMOS) and the parasitic and interconnect capacitances of

the technologies. Presently bipolar devices with a base width of a few tenths of a micron have gate delay of about few hundred picosecond and a speed power product of about 10 picojoule. The I^2L family has a speed power product of few picojoules and a few nanoseconds gate delay. NMOS devices are presently fabricated with a channel length of two microns (4) providing gate delay of about 1 nsec and speed power product of 1 PJoule. CMOS logic with the same channel length as NMOS has about the same gate delay but dissipates less power. The CMOS power dissipation depends on the duty ratio. SOS with its lower parasitic capacitances has about half the speed-power product of CMOS. MESFET devices with about 1 pm gate length have gate delays of about 100 Psec and speed-power products $10 - 100$ Femto-joules. As the average feature size decreases with technology evolution, the performance of all these technologies will improve.

The dimensions of the present NMOS and the smallest conceivable future MOS transistors (5) are shown in Fig. 2. Presently MOS devices have an oxide thickness of 700 \AA and an electrical channel length of $2\mu\text{m}$. Operating from a 5V supply, they provide a gate delay of 1 nanosecond and a speed power product of 1 picojoule.* The smallest conceivable NMOS transistor is projected to have an oxide thickness of about 70 angstrom (limited by tunneling considerations) a channel length of $0.2\mu\text{m}$ (limitied by punch-thru effects and substrate doping fluctutations). It will operate from a power supply of 0.5V ($\approx 20 \text{ KT/e}$) which is the minimum gate voltage swing required to change the device current by several orders of magnitudes.

*The gate delay and speed-power product $P \cdot \tau_d$ of the MOS technology are given by the following relations:

$$\tau_d = F_1 \cdot \tau_f, \quad P \cdot \tau_d = F_2 \cdot (\frac{1}{2} C_G V^2)$$

where τ_f is the time of flight or transit time of the carriers across gate of the MOS transistor, F_1 and F_2 are factors which depend on the gate configuration, fanout and parasitic capacitances, C_G the intrinsic gate capacitance and V the voltage swing. In the depletion load NMOS technology $F_1 \approx F_2 \approx 50-70$ for a fanout of 3.

This device will provide a gate delay of few tens of picoseconds and a speed power product of few femtojoules. At these small dimensions, the MESFET technology may replace the MOSFET technology as it eliminates the gate oxide reliability problems and improves the threshold voltage control.

(6) The base width of the smallest size bipolar device is about 500-700 angstrom (limited by punch-thru effects and doping fluctuations). When the electron transit time across the gate of the MOS transistor becomes comparable to the transit time across the base of the bipolar transistor, both devices will intrinsically provide same current drive and speed performance. The performance difference between the two devices will result from the logic configuration and the parasitic and interconnect capacitances of their technologies.

In Fig. 3, the past and expected future trend of the average feature size is illustrated. The average design rule started at $25\mu\text{m}$ in the early 60's. It has been decreasing at 11% per year and is presently about $3\mu\text{m}$. With a projected average design rule of $0.3\mu\text{m}$ by late 1990's the device density will be of the order of $10^7/\text{cm}^2$. Improvements in the average feature size are achieved by improvements in lithography and pattern etching

technologies. In the 60's, contact printing lithography and negative resist wet etching were used. In the 70's conversion to projection printing with positive resist and plasma etching were made for higher masking yield and better resolution of the etching profiles. For $2\mu\text{m}$ average feature size in the early 80's - projection step and repeat lithography will be required.

(8) For feature size below $1\mu\text{m}$ projected in the late 80's conversion to x-ray step and repeat lithography or electron beam direct writing lithography (8) and reactive ion etching may be needed. The development and capital costs of these lithography and pattern etching technologies are quite enormous. For example in the 60's the costs of a Casper-contact aligner and a wet etching station were about 20K \$ and 4K \$ respectively. In the mid-70's a Perkin Elmer projection aligner costs about 150K\$ and a barrel plasma reactor costs about 40-50K\$. A projection step and repeat aligner costs about 500K\$. The cost of an x-ray step and repeat electron beam direct writing system may exceed \$1 million. Also considerable efforts have to be made to reduce the average defect density as the feature size decreases. In Fig. (4) the past and expected future trend of the chip area for equal production cost is shown. In the past the chip area has been increasing at a rate of about 20% per year, because of the increase in the wafer size from 2" to 3" and the reduction in the average defect density. These were achieved by improving the quality of the masks and the materials, and using cleaner rooms. At this rate the chip area will increase to about few hundred thousand mils in the late 80's.

VLSI MOS DEVICE DESIGN

The improvements in the device parameters of MOS devices by scaling their dimensions (9) are given in Table I. Scaling of device dimensions by a factor

S at constant field to maintain the material reliability results in substantial improvements in the gate delay and speed-power product (energy per switching operation). However the delay required to drive an external capacitance does not scale as favorable as the intrinsic gate delay because of the larger number of stages required to buffer the signal.⁽¹⁰⁾ Note that the power dissipation per unit area remains constant. Thus the power dissipation of logic chips will increase with the chip area which creates difficulties in heat removal from packages. In order to limit the logic chips power dissipation, either the speed or the chip area will have to be reduced. Therefore the power dissipation limitation with scaling will make technologies with lower intrinsic power dissipations, such as CMOS and SOS, more attractive.

It has been verified that with the appropriate scaling of the MOS transistor parameters, its characteristics scale as predicted.⁽⁹⁾ However, not all device parameters can be scaled proportionally due to practical technological constraints. With selective scaling of few device parameters second order device effects dominate the device characteristics as its dimensions reach its physical limits. In Fig.(5) and (6) the strong dependence of the gate voltage threshold and drain punchthrough voltage on the device dimensions and doping profile in the channel and isolation regions are due to the three dimensional field distribution in the device. Accurate device models have to be used for proper device design for VLSI.^(11,12)

The past and expected future trends of the NMOS technology gate delay and speed-power product are illustrated in Fig. (7) and (8). The gate delay has been decreasing very rapidly from 80nsec with p channel metal gate technology in the late 60's to 1nsec with scaled n channel silicon gate depletion load technology⁽⁴⁾ in 1978. The improvement in gate delay has been

achieved with technology and circuit innovations such as silicon gate, n channel, depletion loads, and reduced parasitic capacitances. The gate delay is projected to continue to decrease rapidly as more parasitic capacitances are reduced and transistor parameters are scaled. In the 1990's the decrease in gate delay will slow down, due to electron velocity saturation in the channel and limited reduction of residual parasitic capacitances. The lower limit of gate delay of few tens of psec is more than an order of magnitude lower than present technology gate delay. In Fig.(8) the rapid improvement in speed-power product from 500pJ in late 60's to 1pJ in 1978, is attributed to the decrease in gate delays, parasitic capacitances and power supplies. The reduction in speed-power product is projected to continue reaching a lower limit of few Femtojoule with 0.5V supply.

VLSI CIRCUIT DESIGN:

In Table II the interconnect scaling relations are illustrated assuming constant conductor resistivity and equal scaling of all conductor dimensions by a factor S. This results in a response time and voltage drop which do not scale and a conductor current density which increases up with the scaling factor S. Presently current density in aluminum conductors are limited to about 10^5 amps/cm² to provide adequate reliability against metal migration. These unfavorable scaling properties of the interconnection lines can be reduced by decreasing the conductor resistivity and thickness scaling and development of better alloys. The difference in the scaling properties of the device and the interconnection lines have profound effects on how devices and circuits designs are going to evolve in the future. The fact that the intrinsic device delay scales favorably with device scaling while interconnection line and external capacitance drive delay do not scale, means that technologies

with lower parasitic capacitance and less current drain, such as CMOS and SOS, will become more attractive. This means also that future designs will take special care to minimize interconnection lines (where substantial portion of the delay and energy will be dissipated) and external drives by proper partitioning between the chips and increasing the level of chip integration. Larger integration results in larger chips. In order to maintain the yield and cost per function, redundancy techniques will have to be used.

Redundancy techniques in memory and logic design will have to be used for VLSI, not only to lower the cost per function but also to take advantage of the improvements in the intrinsic device performance with scaling. Redundancy consists of using spare units to replace defective ones so that larger chips with more function complexity and high rates of faults can be used to achieve lower cost per function, better performance and high reliability. Substantial improvements in yield with redundancy have been reported using existing wafer yield models.⁽¹³⁾ Redundancy techniques can be applied at different levels on the system, the wafer or the chip level. In applying redundancy on the system level, the chip profiles are generated and stored in a system directory and the system implements the personalization of the partial chips. Redundancy on the wafer level (wafer scale integration) promises potential improvement in packing density and reliability as the wafer is the final package unit. The implementation of redundancy on the chip and wafer levels involves testing to generate a profile of the nonfunctional areas and personalization to replace the defective areas with functional spare areas. Personalization with non-volatile factory programmable elements (such as laser zapped or fusible links) non-volatile field programmable storage elements (such as MNOS and FAMOS) and on-chip latches have been proposed.

The complexity level of circuit design for VLSI requires a new type of design methodology. Design of logic chips with 100K devices and more takes more than 50 men-years. Management of such complex designs will require coordination between the functional definition, the architectural description, the logic interpretation, the circuit design, the physical layout design, the wafer fabrication and testing with verification and validation at each of these levels. This requires proper partitioning of the design into small blocks and macros of manageable sizes whose design can easily evolve with technology advancement. In order to make the testing of these complex chips feasible, testability have to be designed in by providing means to test inaccessible nodes in the circuits, structuring the design into independently testable blocks and/or incorporating circuits to perform self-checking routines. The implementation of these complex designs will rely on the use of computer aids and interactive graphics for the simulation, design and verification at the different phases of the design.

FUTURE TRENDS

Since the integrated circuit revolution started in the early 60's, the number of components per chip have increased by about five orders of magnitudes.⁽¹⁾ Today memory devices with 64K bits per chip and 16 bit microprocessors with 30K devices per chip are becoming available. Projecting that increase in the future until the devices reach their physical limits indicate that complexity of few hundred million components per chip will be reached by the end of the century. This is more than three orders of magnitude increase in component complexity per chip.

The rate of technology development in the future will be limited by the ability to model, design and fabricate devices with dimensions approaching its physical limits. The interconnection lines delay and current density, the device second order effects and the chip power dissipation will limit the technology performance. Therefore complementary and low parasitic capacitance technologies will become more attractive. It is also quite likely that the MESFET technology will replace the MOSFET technology as device dimensions reach the physical limits due to higher reliability and better threshold voltage stability.

The complex design and product definition in VLSI will limit the component complexity on logic chips. The development of programmable function logic chips, such as microprocessors and control chips, will follow two paths. One path will follow present architecture with more CPU, memory and I/O circuits running at a faster rate and a better user interface. By 1990's logic chips of few million gates complexity will run at 100MHz from 0.5V supply dissipating few watts. In the other path, new architectures will be developed to provide better matching between the silicon technology and its topological properties and the function architecture software and human interface. The important topological properties of the VLSI silicon technology are the facts that a substantial part of the delay and energy is dissipated in communication across the interconnection lines and that both memory and logic are built with the same technology.

The memory device design will continue to lead the technology development. Denser memory cells will be made with scaled devices and more complex vertical structures. For optimum speed, signal sensing margins and testing considerations, memories will be organized in blocks with more emphasis on redundancy.

Memory design will also follow two paths. One path will follow present function organization with more memory running at a faster access time and lower power per bit. In the other path new organizations with smarter functions on the chip will be developed.

CONCLUSIONS

VLSI offers tremendous challenges in product definition, circuit design, device modeling and design, technology developments and capital investments. The complexity of product definition will limit the component counts on system function logic chips. The complex circuit designs will require hierarchical structured design methodology with considerations to testability heavy reliance on computer design aids and more emphasis on redundancy. The interconnection line delay and current density, the device second order effects and chip power dissipation will limit the technology performance.

REFERENCES

1. G. Moore, "Progress in Digital Electronics", IEDM Digest of Technical Papers, December 1975, pp 13.
2. J.H. Yvan and E. Harari, "Short Channel CMOS/SOS Technology", IEEE Transactions on Electron Devices, Vol. ED-25, No 8, August 1978, pp 989-995.
3. R. Eden, "GaAs Integrated Circuits - MSI Status and VLSI Prospects". IEDM Digest of Technical Papers, December 1978, pp 6-11.
4. R. Pashley et al, "A High Performance 4K Static RAM Fabricated With an Advanced MOS Technology", ISSCC Digest of Technical Papers, pp 22-23, February 1977.
5. B. Honeisen and C.A. Mead, "Fundamental Limitations in Microelectronics": I MOS Technology:, Solid State Electronics, Vol. 12, 1972, pp 819-829.
6. H.M. Dareley et at, "Fabrication and Performance of Submicron Silicon MESFET", IEDM Digest of Technical Papers, December 1978, pp62-65.
7. B. Honeisen and C.A. Mead, "Fundamental Limitations in Microelectronics, II Bipolar Technology", Solid State Electronics, Vol. 15, 1972, pp 891-897.
8. A. Broers, "Fine Line Lithography Systems for VLSI", IEDM Digest of Technical Papers, December 1978, pp 1-5.
9. R.H. Dennard et al, "Design of Ion Implanted MOSFET's With Very Small Physical Dimensions", IEEE Journal of Solid State Circuits, Vol. SC-9 October 1974.
10. A. Mohsen and C. Mead, "Optimization of Driving and Sensing of Signals on High Capacitance Paths for VLSI", IEEE Journal of Solid State Circuits, April 1979.
11. P. Wang, "Device Characteristics of Short-Channel and Narrow Width MOSFET's" IEEE Transactions on Electron Devices, Vol. ED-25, July 1975, pp 779-786.

REFERENCES

12. S. Shimohigashi et al, "Characteristics of Short Channel MOSFETS in the Punchthrough Current Mode", IEDM Digest of Technical Papers, December 1978, pp 66-68.
13. S.E. Shuster "Multiple Word/Bit Line Redundancy for Semiconductor Memories", IEEE Journal of Solid State Circuits, Vol. SC-13, October 1978, pp 698-702.

TABLE I

FET DEVICE SCALING FOR CONSTANT FIELD

<u>DEVICE/CIRCUIT PARAMETRE</u>	<u>SCALING FACTOR</u>
DEVICE DIMENSIONS L, W, TOX	1/S
DOPING CONCENTRATION	S
VOLTAGE	1/S
FIELD	1
CURRENT	1/S
DEVICE EQUIVALENT RESISTANCE	1
GATE DELAY	1/S
EXTERNAL CAPACITANCE DRIVE	1/S, LNS
POWER DISSIPATION/DEVICE	1/S ²
POWER DENSITY	1
SPEED POWER PRODUCT/DEVICE	1/S ³

TABLE II
INTERCONNECT SCALING RELATIONS

<u>PARAMETRE</u>	<u>SCALING FACTOR</u>
LINE RESISTANCE $R_L = \rho L / Wd$	S
LINE CAPACITANCE $C_L = \epsilon L W / D$	1/S
LINE RESPONSE TIME $R_L C_L$	1
LINE VOLTAGE DROP $I R_L$	1
LINE CURRENT DENSITY I / Wd	S

FIGURE CAPTIONS

Figure (1) Performance of various technologies in 1978.

Figure (2) The 1978 NMOS transistor and the 1990's smallest size MOS transistor.

Figure (3) Past and projected future trend of average design rule.

Figure (4) Past and projected future trend of MOS LSI die size for equal manufacturing cost.

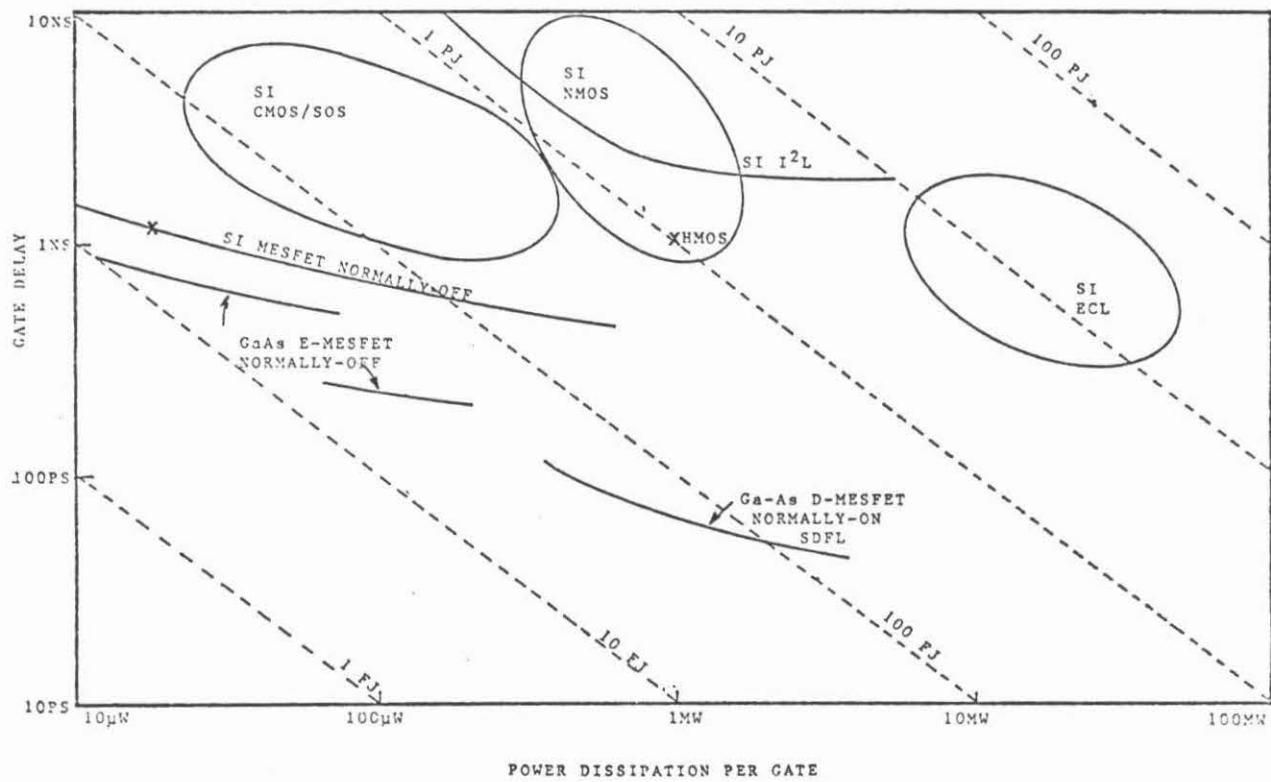
Figure (5) The dependence of gate threshold voltage on the device geometry and doping profile.

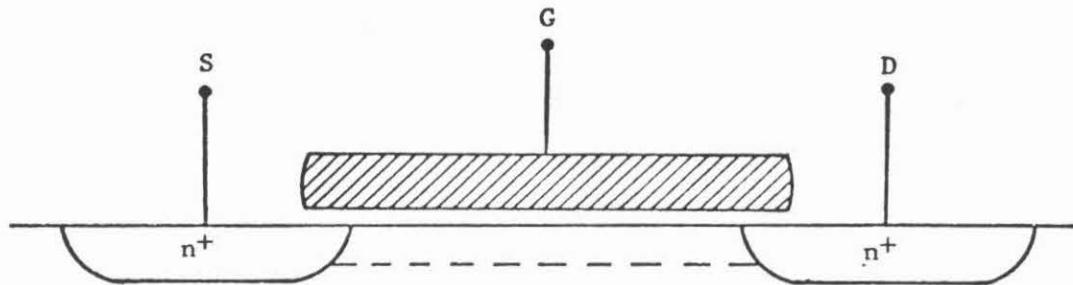
Figure (6) The dependence of drain punchthrough voltage on the device geometry and doping profile.

Figure (7) Past and projected future trend of NMOS technology gate delay.

Figure (8) Past and projected future trend of NMOS technology speed-power product.

SPEED POWER PRODUCT OF VARIOUS TECHNOLOGIES AT 1978

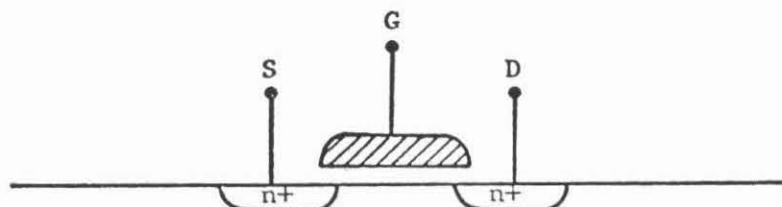




1978 NMOS

$$L_{CH} = 2\mu M, \tau_{OX} = 700A^0, X_J = 0.5\mu M$$

$$\tau_F = 15PSEC, \tau_D = 1NSEC, P\tau_D = 1PJ, V_{CC} = 5V$$

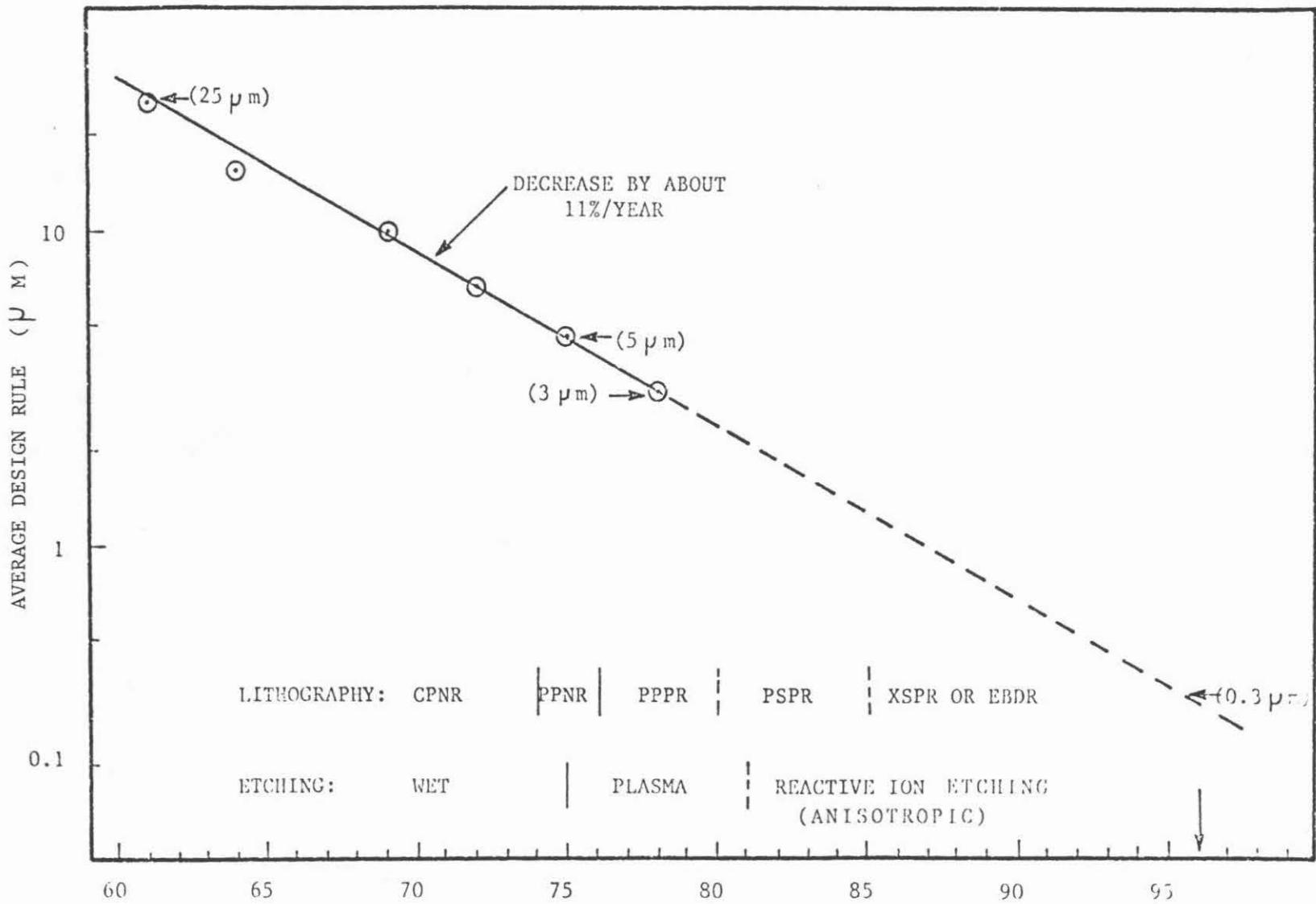


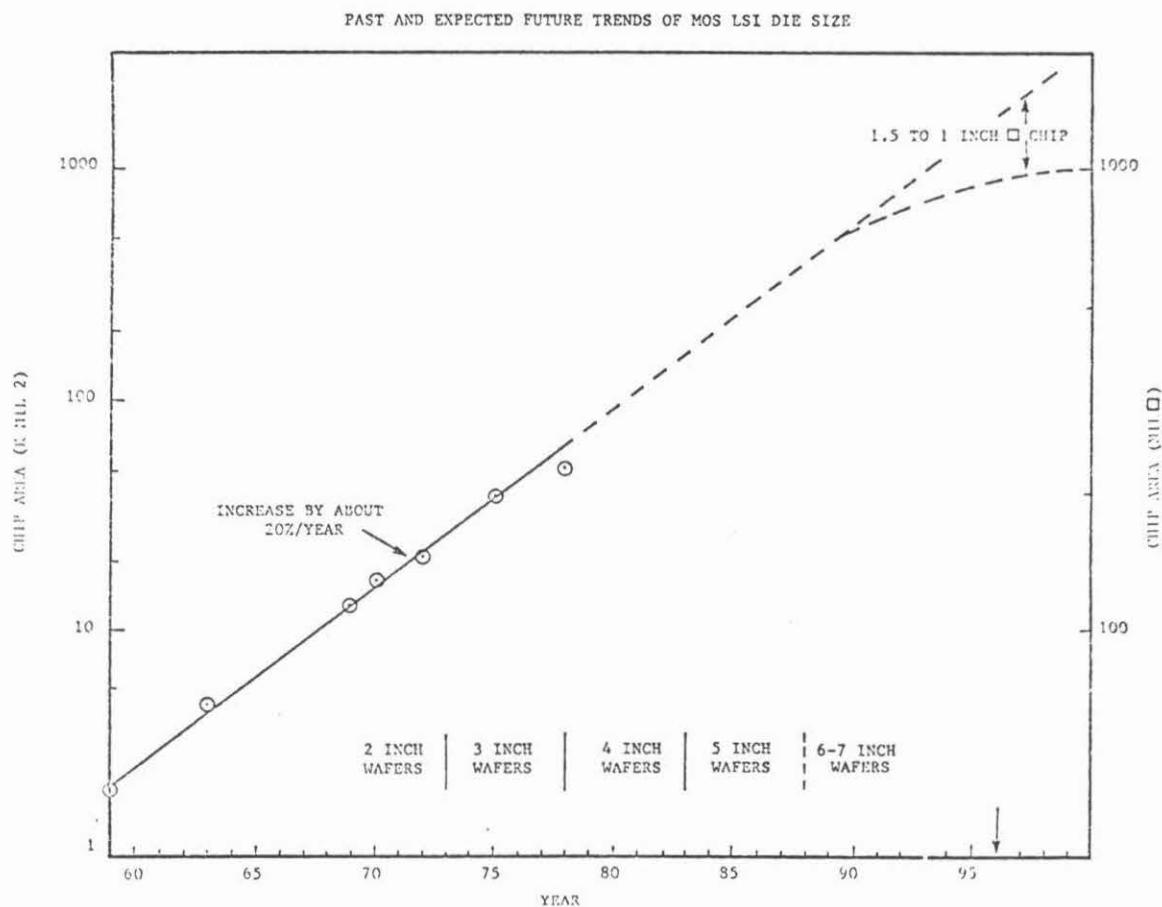
1990's NMOS

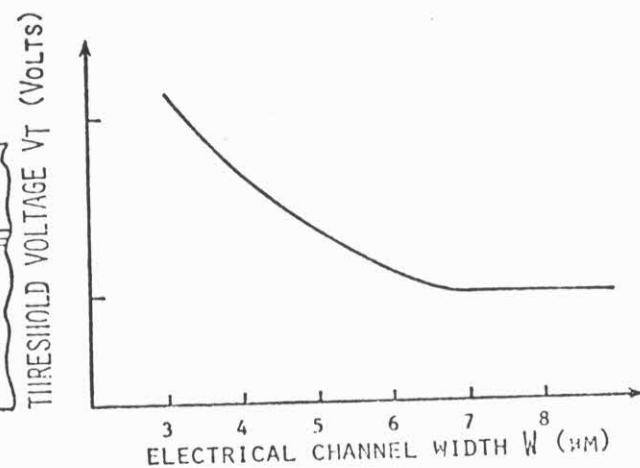
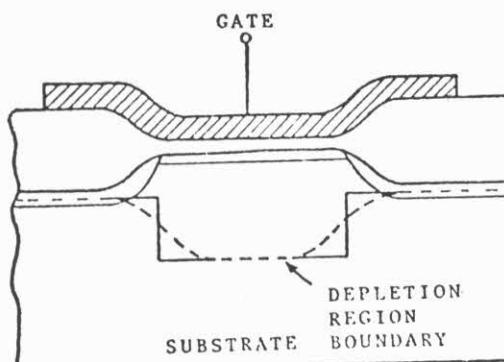
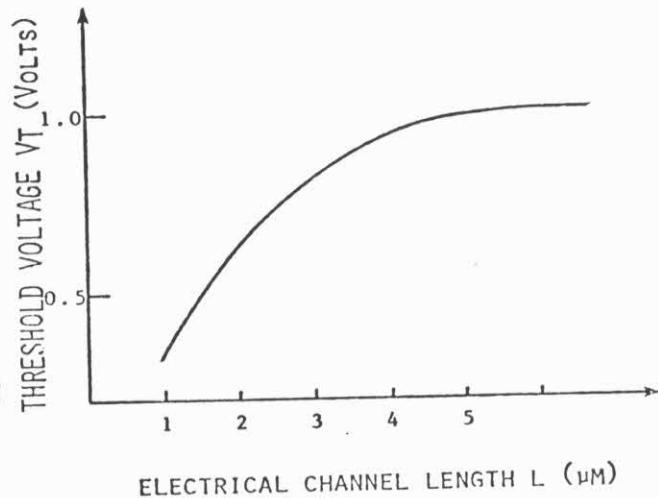
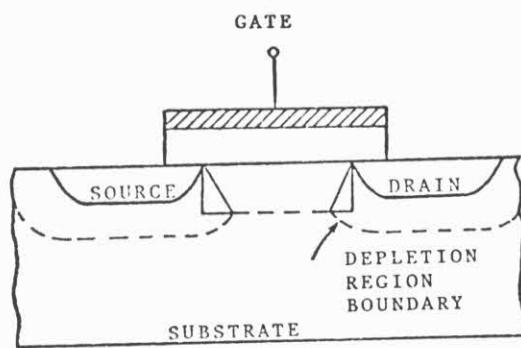
$$L_{CH} = 0.2\mu M, \tau_{OX} = 70A^0, X_J = 0.05\mu M$$

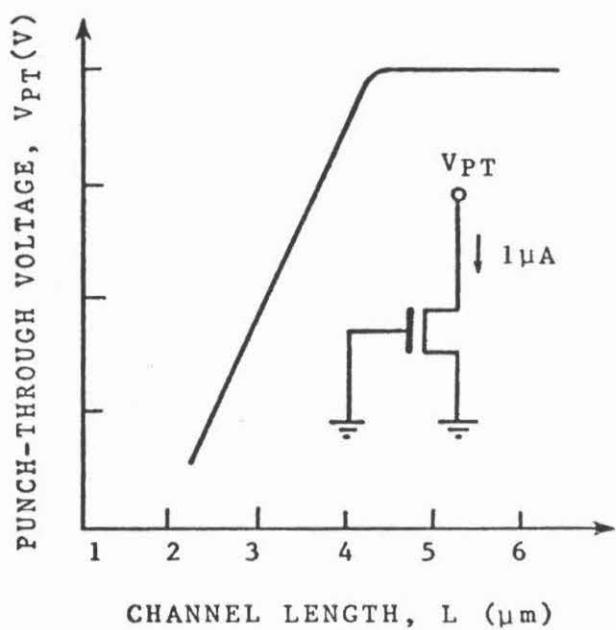
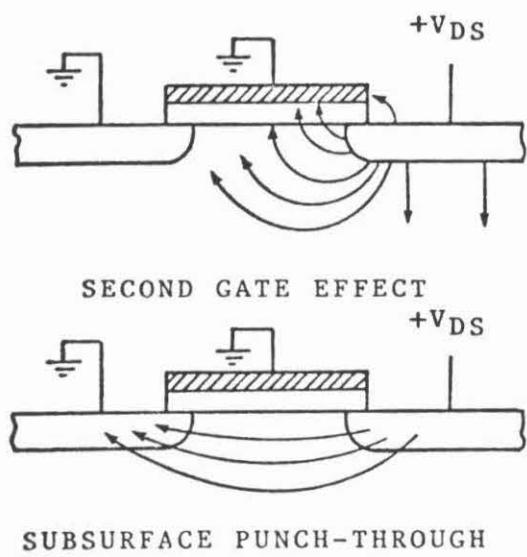
$$\tau_F = 1 \text{ PSEC}, \tau_D = 50PSEC, P\tau_D = 2FJ, V_{CC} = 0.5V$$

PAST AND EXPECTED FUTURE TREND OF AVERAGE DESIGN RULE

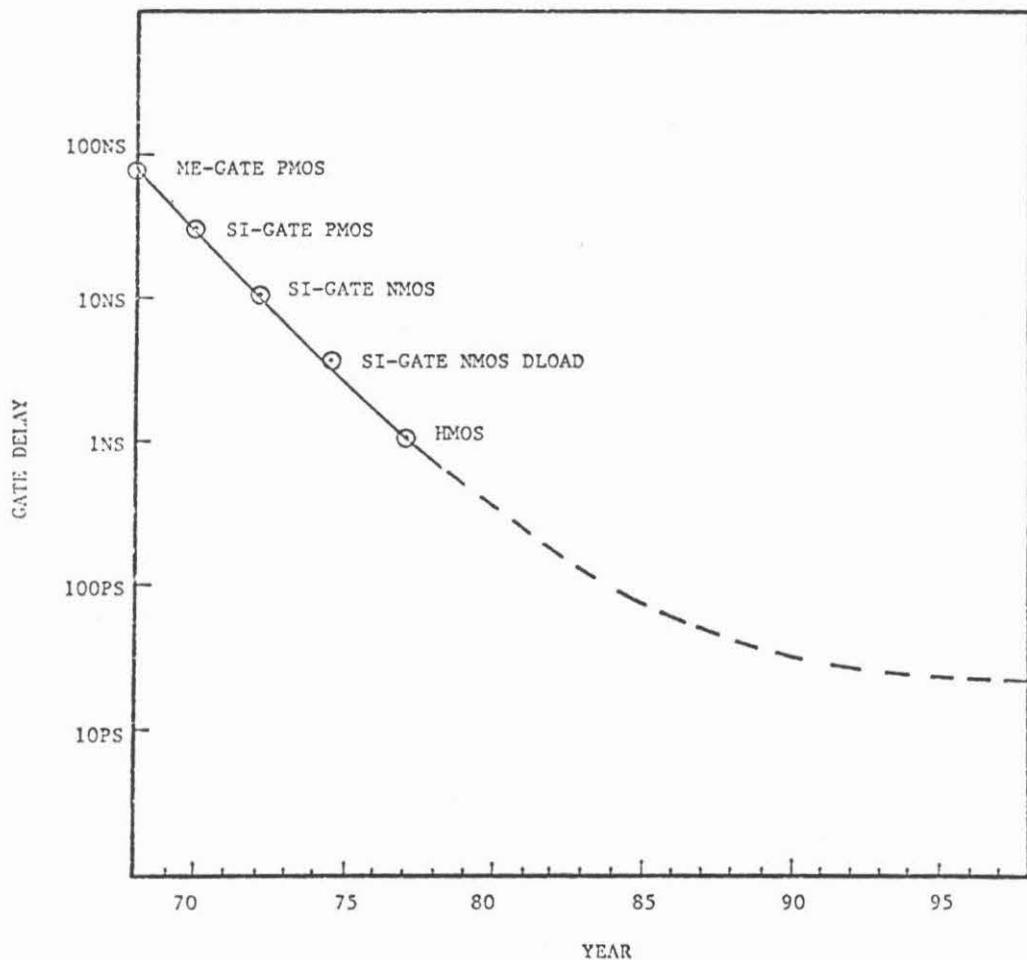




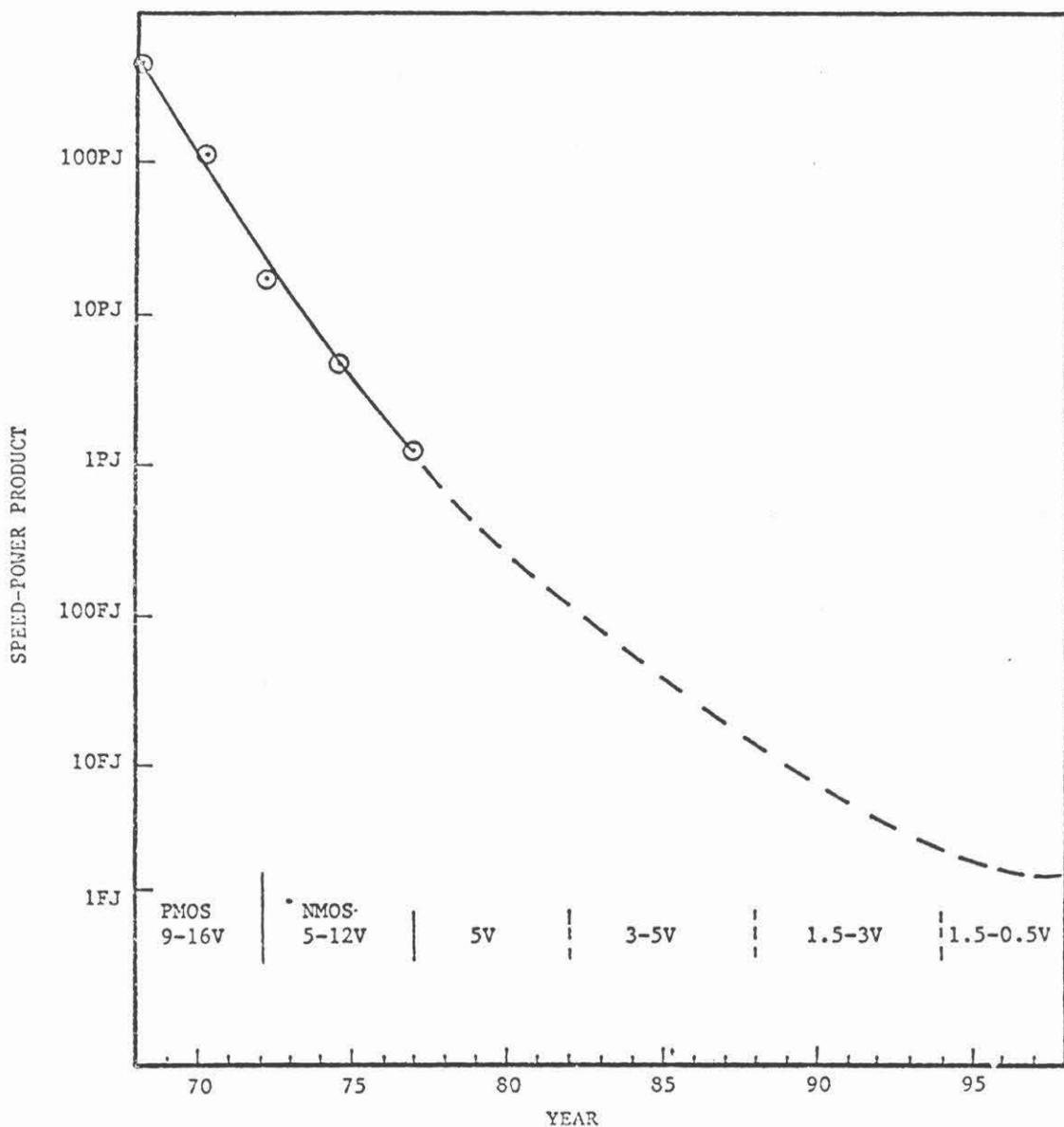




PAST AND EXPECTED FUTURE TREND OF MOS TECHNOLOGY
GATE DELAY



PAST AND EXPECTED FUTURE TREND OF MOS TECHNOLOGY SPEED-POWER PRODUCT



MATHEMATICAL ASPECTS OF VLSI DESIGN

Martin Rem

Eindhoven University of Technology

and

California Institute of Technology

```

0 0 1 0   1 1 0 1   0 0 0 1
1 0 0 0   0 0 0 0   0 1 0 0
0 0 1 1   1 0 1 0   1 0 0 1
0 0 1 1   0 0 1 0   1 1 1 1
1 1 0 1   1 1 0 1   1 0 0 0
1 1 1 0   1 1 0 1   0 1 0 1
0 0 1 0   0 0 1 1   1 1 1 0

```

The above is a computer program, written in the way we would program in the fifties. The program is represented in the very same way it is stored in the computer: it is the lowest level description of a program. Nowadays a program will be written in a notation more like the following.

```

do x > y → x:= x - y
  | y > x → y:= y - x
od

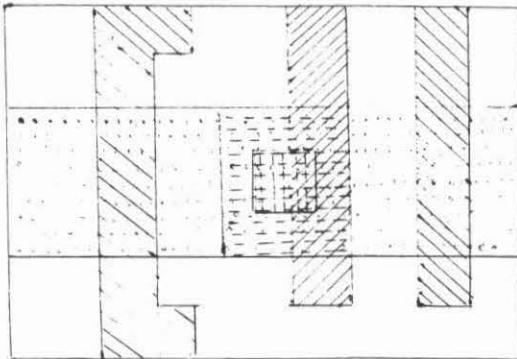
```

Although the program is written in a modern notation the algorithm it expresses is quite old. It actually dates back to the Greeks: it is Euclid's algorithm to determine the greatest common divisor of two numbers, x and y in this case. We can write our programs in such a clear way because we can make compilers that transform them into the required binary code. We don't want to know what binary code is produced, we don't even wish to know the binary code. Nor do we want the compiler to generate any messages that refer to the binary code. It should behave as if the above text is directly executed and all (error)messages should be phrased in terms of that program text.

There is more to it than just clarity. The program is expressed in well-defined constructs, each construct having a well-defined meaning. That

allows us to prove properties of our programs by which we can gain understanding in what is involved in programming and by which we can raise the confidence level of our products (cf. [2]). This enables us to construct larger programs or systems in a correct way. These larger systems will then consist of a hierarchy of smaller systems.

This is all very well-known, but let us now look at the following picture.



The above is a chip layout, or at least part of it. It also expresses some computing system, a program if you like. As in the case of the binary code it is represented in the same way it can actually be found in the computer. It is again the lowest level description of a system. But it is still the level at which we design. You may actually encounter designers drawing these figures with colour pencils on large sheets of paper. In a more modern environment you may find television screens drawing the figures for them, but it is in terms of these pictures that the designer understands his system.

The moral of this observation should be clear. We wish to have an algorithmic notation for computing structures in which we can express what should happen rather than how it should happen, together with a compiler that, if we so desire, can generate the chip layout: a silicon compiler.

+ + +

There are a number of respects in which chips are new. I want to mention three of them.

(1) New balance between logic and communication.

VLSI provides a homogeneous medium in which both logic and storage can be realized. The transistors come almost for free, it is the "wires", the communication, that determine the cost, both in area and in speed, of the chip [8]. The consequence is that traditional switching theory and complexity theory are not directly applicable to VLSI design as they don't take communication requirements into account.

(2) Invitation to high concurrency.

The expensive communication and the uniform technology form an invitation to introduce many local computations that are executed concurrently and that jointly carry out the required computation. The idea is to do the operations where the arguments are, rather than shipping the arguments to processing units.

The design of such an ultraconcurrent computation is not an easy task. In this respect VLSI came too early: we are beginning to understand the theory of sequential programming, but we still have only a rudimentary knowledge of concurrent programming.

(3) Geometrical composition of constructs.

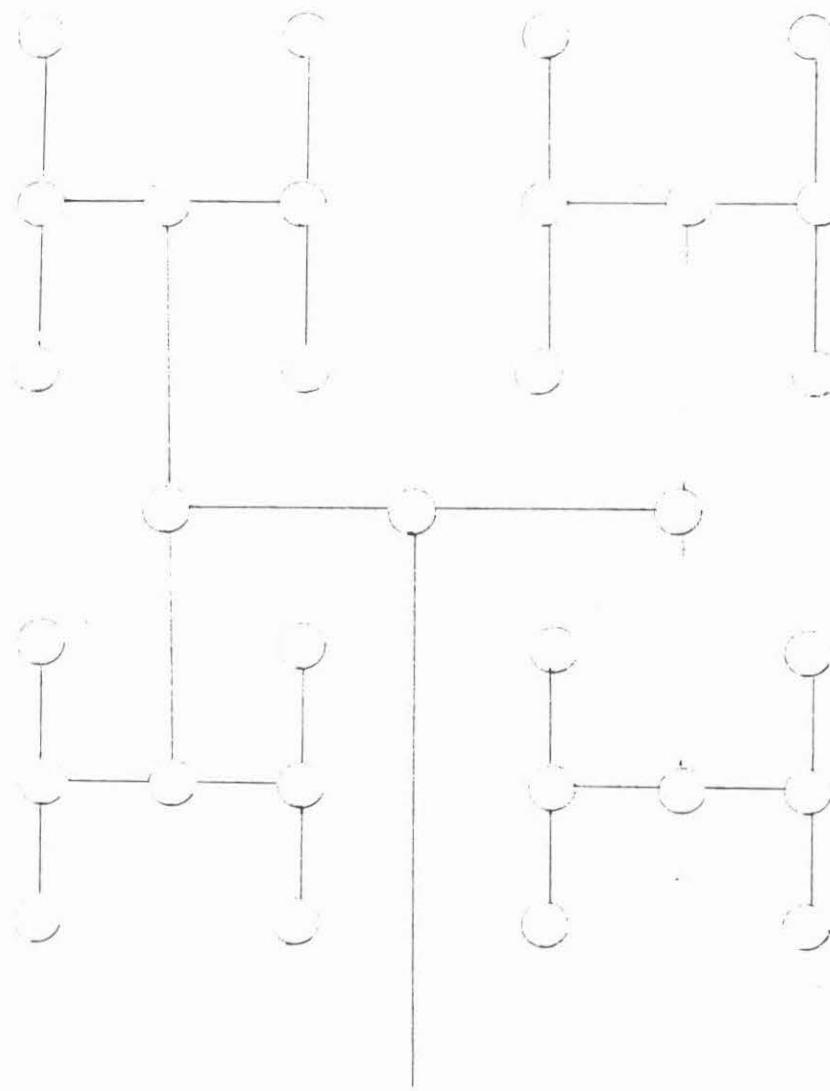
In programming we are used to think in terms of functional composition of constructs, compositions like recursion and the constructs of structured programming. We are beginning to understand them, but now there comes an additional constraint: the structures have to be mappable unto the plane. The structures should, therefore, be regular and they somehow must "fit". Ideally, they should resemble those plane-covering drawings by M.C. Escher.

The number of regular structures is limited. I shall mention some of them, more or less in the order of decreasing mappability unto the plane.

- a. vector (pipe line)
- b. ring
- c. matrix
- d. binary tree

In a. the maximal distance (in number of connections) between any two ele-

ments is n , or $n - 1$, for n elements. In b. it is $n/2$, in c. \sqrt{n} , and in d. $\log_2 n$. One may wonder whether the tree is mappable unto the plane. It turns out that that is not too bad.



The above picture must remind one of an Escher drawing. It is a binary tree of 5 levels and hence 31 nodes, 16 of them being leaves. Notice that the arcs get longer towards the root. That is fortunate as most of the arcs are at the lower levels, i.e. towards the leaves of the tree.

e. boolean k-cube

Let n be 2^k , number the elements 0 through $n - 1$ and write every element number in binary notation. An element is connected to every other element whose number is at Hamming distance 1, i.e. differs in one bit.

Every element has, consequently, $\log_2 n$ ($=k$) neighbours. This scheme is still mappable unto the plane, although not as well as the binary tree. Like the binary tree it has a maximal distance of $\log_2 n$. An important difference with the binary tree is that the cube does not have a designated root and this may prevent the congestion problems that may very well occur at the tree's root. However, the fact that the number of neighbours depends on the total number of elements is an awkward property: it precludes the modular composition of such a network. A scheme that looks like the cube without having this property is the following one.

f. perfect shuffle

Again the element numbers are coded in k ($=\log_2 n$) bits, but now an element is connected to four neighbours, viz. those with which it has $k - 1$ consecutive bits, i.e. all but the first all or all but the last bit, in common. Again the maximal distance between any two elements is $\log_2 n$. All nodes are equivalent, a property it shares with the cube. The problem with the perfect shuffle is that I don't know how mappable it is unto the plane. I have my doubts there.

+ + +

One of the problems with chips, nowadays, is that their initial design costs are very high. The consequence is that only those chips are produced for which a large market is expected. This phenomenon, of course, impedes progress. The hope is that the advent of the silicon compiler mentioned earlier will resolve this unfortunate situation. It will then become feasible to build small quantities of special purpose chips.

More interesting is the design of highly concurrent general purpose computing engines. In such a machine the computing elements will be connected by some pattern of "wires". How does one program such a machine? Does the programmer map his computation explicitly unto the connection pattern provided? Or does the programmer use an algorithmic notation, a programming language if you prefer, that guarantees the mappability of his computation unto the connection pattern? It seems that the latter solution is to be preferred.

The ideal is to have a uniform notation for computations. From the notation

one should not be able to tell whether the computation is meant to be

- a chip layout
- a computation for a graph of communicating processes like, e.g.,
a tree machine, or
- a computation for a sequential machine.

The only thing expressed by the notation is the computation and there should be compilers for all three realizations above.

An important problem is finding such a uniform notation for computations. A number of proposals have emerged recently. To mention just a few of them:

- Actors [4]
- Assocs [7]
- Data driven nets [1]
- Communicating sequential processes [5]

The latter one is a rather nice notation and I would like to show an example of it. This particular one was written by David Gries. It expresses a computation of all primes less than 10000, using 102 processes and a print process. The code of the print process is not shown. It is again an old algorithm: the sieve of Eratosthenes.

The notation is a blend of Dijkstra's guarded commands [3] and synchronized communication. The sending and receiving of data are represented by an exclamation point and a question mark respectively. A matching pair of communication commands, one in the sending process and one in the receiving process, is executed simultaneously.

Each process SIEVE(*i*) sends one prime (the *i*-th prime) to the print process and sieves all multiples of that prime out of the stream of numbers it receives from process SIEVE(*i*-1). The stream is generated by process SIEVE(0).

```

SIEVE(i: 1 .. 100)::

    SIEVE(i-1)?p; PRINT!p;

    mp:= p;
    do SIEVE(i-1)?m →
        do m > mp → mp:= mp + p od;
        if m = mp → skip
        || m < mp → SIEVE(i+1)!m
    fi
od

SIEVE(0)::

    PRINT!2;
    m:= 3;
    do m < 10000 → SIEVE(1)!m; m:= m + 2 od

SIEVE(101)::

    SIEVE(100)?p; PRINT!p

```

What all these proposals lack is the notion of local computation. Every process, actor, or net element may in principle communicate with every other. To remedy this I propose the following concept of hierarchical processes, that takes locality into account.

A process consists of a program, a state space, an initial state, and a (possibly empty) set of subprocesses.

This is a recursive definition defining a hierarchy of processes. It thus maps naturally unto a tree. The state space is the set of all possible states of the process. The program consists of sequencing primitives and instructions. Only instructions can change the state of the process. Analogous to [5] communication is performed pairwise synchronized. If P is a subprocess of Q then Q is called the environment of P . Two processes having the same environment are called coprocesses. Communication may take place only between coprocesses or between a process and its environment. The proposal is, therefore, more general than just a tree. In a tree we only have communication between a process and its environment. I am proposing to allow "horizontal" communication between processes with the same environment as well. The hope is that this more

general scheme resolves the congestion problem at roots and is still restricted enough to be in general well-mappable unto the plane.

A process' instruction is either private or public. All non-communicating instructions and communicating instructions with subprocesses are private instructions of the process. (The latter ones are public for the subprocesses.) Communicating instructions with coprocesses or with the environment of a process are public instructions. (The latter ones are private for the environment.)

A hierarchy is the only way to build complex systems with a high confidence level. They enjoy the nice property that we can prove assertions about the system by recursion over the hierarchy: assuming that the assertion holds for the subprocesses we prove that it holds for the process itself as well. During this proof we don't look inside the subprocesses, we only use their public instructions. Nor do we look at the coprocesses or the environment of the process, we only use the process' private instructions. That seems to be the only way to keep complex structures understandable.

We are as a matter of fact quite lucky here. There are physical reasons [6] why we want to design hierarchical systems, but also because of properties like understandability and inherent simplicity we wish to have hierarchies. What is mathematically attractive turns out to be physically attractive as well. Is this a violation of Murphy's law?

References

- [1] Davis, A.L. "A maximally concurrent, procedural, parallel process representation". University of Utah, Salt Lake City, Utah, 1978.
- [2] Dijkstra, Edsger W. "A Discipline of Programming". Prentice-Hall, Englewood Cliffs, N.J., 1976.
- [3] Dijkstra, Edsger W. "Guarded commands, nondeterminacy and formal derivation of programs". Comm. ACM 18,8 (August 1975), 453-457.
- [4] Hewitt, Carl & Henry Baker. "Laws for communicating parallel processes". Information Processing 77, North-Holland, Amsterdam, 1977, 987-992.

- [5] Hoare, C.A.R. "Communicating sequential processes". Comm. ACM 21,8 (August 1978), 666-677.
- [6] Mead, Carver A. & Martin Rem. "Cost and performance of VLSI computing structures". Japan-USA Computer Conference, San Francisco, 1978.
- [7] Rem, Martin. "Associations and the Closure Statement". MC Tract 76, Mathematical Centre, Amsterdam, 1976.
- [8] Sutherland, Ivan E. & Carver A. Mead. "Microelectronics and computer science". Scientific American 237,3 (September 1977), 210-228.

Let's Design Algorithms for VLSI Systems

H. T. Kung

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

January 1979

This research is supported in part by the National Science Foundation under Grant MCS 75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422.

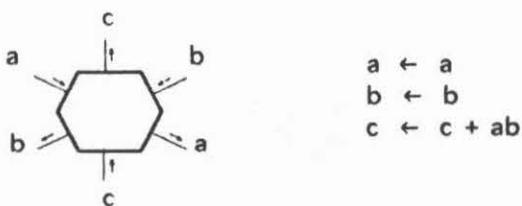
1. Introduction

Very Large Scale Integration (VLSI) technology offers the potential of implementing complex algorithms directly in hardware [Mead and Conway 79]. This paper (i) gives examples of algorithms that we believe are suitable for VLSI implementation, (ii) provides a taxonomy for algorithms based on their communication structures, and (iii) discusses some of the insights that are beginning to emerge from our efforts in designing algorithms for VLSI systems.

To illustrate the kind of algorithms in which we are interested, we first review, in Section 2, the matrix multiplication algorithm in [Kung and Leiserson 78] which uses the hexagonal array as its communication geometry. In Section 3, we discuss issues in the design of VLSI algorithms, and classify algorithms according to their communication geometries. Sections 4 to 7 represent an attempt to characterize computations that match various processor interconnection schemes. Special attention is paid to the linear array connection, since it is the simplest communication structure to build and is fundamental to other structures. Some concluding remarks are given in the last section.

2. A Hexagonal Processor Array for Matrix Multiplication --- An Example

Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ band matrices with band width w_1 and w_2 , respectively. Their product $C = (c_{ij})$ can be computed in $3n + \min(w_1, w_2)$ units of time by an array of $w_1 w_2$ hexagonally connected "inner product step processors". Note that computing C on a uniprocessor using the standard algorithm would require time proportional to $O(w_1 w_2 n)$. As shown in Figure 1, an inner product step processor updates c by $c \leftarrow c + ab$ and passes data a, b at each cycle.



We illustrate the computation on the hexagonal array by considering the band matrix multiplication problem in Figure 2.

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ a_{41} & & \cdot & & \\ 0 & & \cdot & & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & & \cdot & & \\ 0 & & \cdot & & \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & \\ c_{31} & c_{32} & c_{33} & c_{34} & \\ c_{41} & c_{42} & & \cdot & \\ 0 & & & \cdot & \end{bmatrix}$$

Figure 2: Band matrix multiplication.

The diamond shaped hexagonal array for this case is shown in Figure 3, where arrows indicate the direction of the data flow. The elements in the bands of A, B and C march synchronously through the network in three directions. Each c_{ij} is initialized to zero as it enters the network through the bottom boundaries. (For the general problem of computing $C=AB+D$ where $D=(d_{ij})$ is any given matrix, each c_{ij} should be initialized to the corresponding d_{ij} .) One can easily see that each c_{ij} is able to accumulate all its terms before it leaves the network through the upper boundaries.

3. The Structure of VLSI Algorithms

3.1. Three Attributes of a VLSI Algorithm

There are three important attributes of the matrix multiplication algorithm described in the preceding section, or of any VLSI algorithm in general. In the following, we discuss these attributes. We also suggest how an algorithm well-suited for VLSI implementation will appear in terms of these attributes.

Function of each processor

A processor may perform any constant-time operation such as an inner product step, a comparison-exchange, or simply a passage of data. For implementation reasons, it is desirable that the logic and storage requirement at each processor be as small as possible

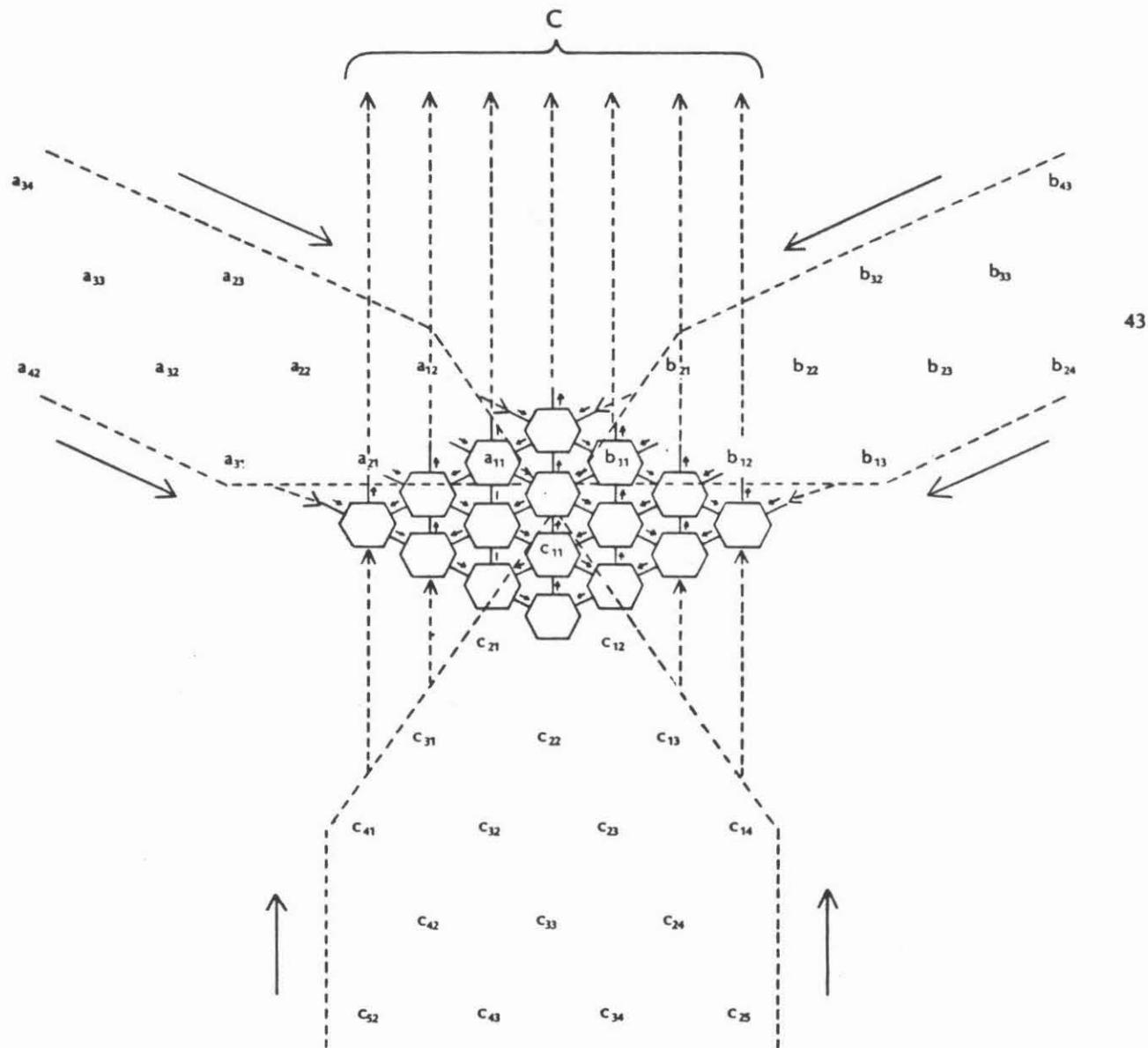


Figure 3: The hexagonal array for the matrix multiplication problem in Figure 2.

and that the majority of processors be uniform. The processors that communicate with the outside world are of course special. The number of these special I/O processors should be kept as small as possible because of pin constraints.

Communication Geometry

The processors in the matrix multiplication algorithm communicate with each other through a hexagonal array network. The communication geometry of a VLSI algorithm refers to the geometrical arrangement of its underlying network. Chip area, time, and

power required for implementing an algorithm are largely dominated by the communication geometry of the algorithm [Sutherland and Mead 77]. It is essential that the geometry of an algorithm be simple and regular because such a geometry leads to high density and, more importantly, to modular design. There are few communication geometries which are truly simple and regular. For example, there are only three regular figures -- the square, the hexagon, and the equilateral triangle -- which will close pack to completely cover a two-dimensional area. The remainder of the paper deals mainly with algorithms with simple and regular communication geometries.

Data Movement

The manner in which data circulates on the underlying network of processors is a critical aspect of a VLSI algorithm. Pipelining, a form of computation frequently used in VLSI algorithms, is an example of data movement. Conceptually, it is convenient to think of data as moving synchronously, although asynchronous implementations may sometimes be more attractive. Data movement is characterized in at least the following three dimensions: direction, speed, and timing. An algorithm can involve data being transmitted in different directions at different speeds. The timing refers to how data items in a data stream should be configured so that the right data will reach the right place at the right time. Consider, for example, the matrix multiplication algorithm in Figure 3. There are three data streams, consisting of entries in matrices A, B, and C. The data streams move at the same speed in three directions, and elements in each diagonal of a matrix are separated by three time units. To reduce the complexity in control, it is important that data movements be simple, regular, and uniform.

3.2. Systolic Systems

It is instructive to view a VLSI algorithm as a circulatory system where the function of a processor is analogous to that of the heart. Every processor rhythmically pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network. In [Kung and Leiserson 78], a network of (identical) simple processors that circulate data in a regular fashion is called a systolic system. (The word "systole", borrowed from physiologists, originally refers to the recurrent contractions of the heart and arteries which pulse blood through the body.) Systolic computations are characterized by the strong emphasis upon data movement, pipelining in particular. VLSI algorithms are examples of systolic systems.

3.3. A Taxonomy for VLSI algorithms

We give a taxonomy for VLSI algorithms based on their communication geometries. This taxonomy provides a framework for characterizing computations on the basis of their communication structures. The table on the next page provides examples of algorithms classified by the taxonomy. Most of these algorithms will be discussed in subsequent sections of this paper.

4. Algorithms Using One-dimensional Linear Arrays

One-dimensional linear arrays represent the simplest way of connecting processors (see Figure 4). It is important to understand the characteristics of this simplest geometry, since it is the easiest connection scheme to build and is the basis for other communication geometries.



Figure 4: A one-dimensional linear array.

The main characteristic of the linear array geometry is that it can be viewed as a pipe and thus is natural for pipelined computations. Depending on the algorithm, data may flow in only one direction or in both directions simultaneously.

4.1. One-way Pipeline Algorithms

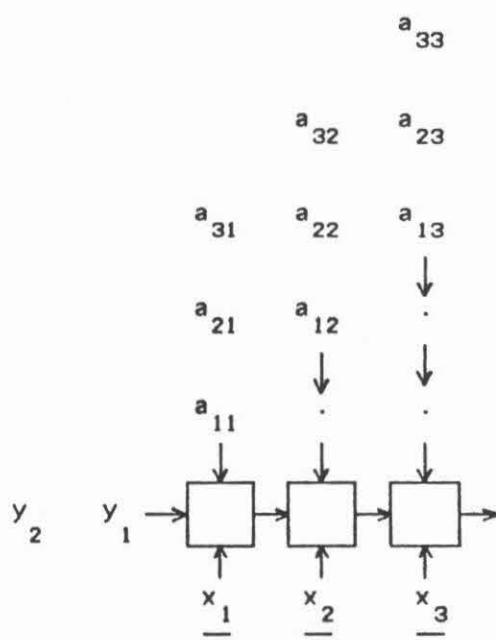
One-way pipeline algorithms correspond to the classical concept of pipeline computations [Chen 75]. That is, results are formed (or "assembled") as they travel through the pipe (or "the assembly line") in one direction. Matrix-vector multiplication is a typical example of those problems that can be solved by one-way pipeline algorithms. For example, the matrix-vector multiplication in Figure 5 (a) can be pipelined using a set of linearly connected inner product step processors. Referring to Figure 6, an inner product step processor, similar to that in Figure 1, updates y by $y \leftarrow y + ax$ at each cycle. Figure 5 (b) illustrates the timing of the pipeline computation. In a synchronous manner, the a_{ij} 's march down and the y_i 's, initialized as zeros, march to the right. The y_1 accumulates its first,

Examples of VLSI Algorithms

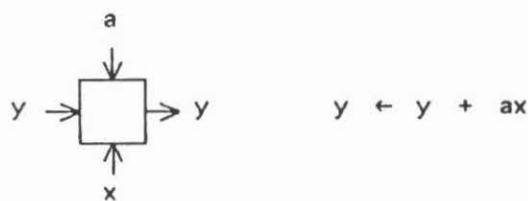
Communication Geometry	Examples
1-dim linear arrays	Matrix-vector multiplication FIR filter Convolution DFT Carry pipelining Pipeline arithmetic units Real-time recurrence evaluation Solution of triangular linear systems Constant-time priority queue, on-line sort Cartesian product Odd-even transposition sort
2-dim square arrays	Dynamic programming for optimal parenthesization Numerical relaxation for PDE Merge sort FFT Graph algorithms using adjacency matrices
2-dim hexagonal arrays	Matrix multiplication Transitive closure LU-decomposition by Gaussian elimination without pivoting
Trees	Searching algorithms Queries on nearest neighbor, rank, etc. NP-complete problems systolic search tree Parallel function evaluation Recurrence evaluation
Shuffle-exchange networks	FFT Bitonic sort

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ \vdots & \vdots & \vdots \\ a_{51} & a_{52} & a_{53} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ \vdots \end{bmatrix}$$

(a)



(b)

Figure 5: (a) Matrix-vector multiplication and (b) one-way pipeline computation.**Figure 6:** The inner product step processor for the linear array in Figure 5 (b).

second, and third terms at time 1, 2, and 3, respectively, whereas the y_2 accumulates its

first, second, and third terms at time 2, 3, and 4, respectively. Thus, this is a (left-to-right) one-way pipeline computation. In the figure, the x_i 's are underlined to denote the fact that the same x_i is fed into the processor at each step in the computation (so x_i can actually be a constant stored in the processor). This notation will be used throughout the paper.

Any problem involving a set of independent multi-stage computations of the same type can be viewed as a matrix-vector multiplication. That is, each independent computation corresponds to the computation of a component in the resulting vector, and each stage of the computation corresponds to an "inner product step" of the form $y \leftarrow F(a, x, y)$ for some function F . Consequently, with linearly connected processors capable of performing these functions F , the problem can be solved rapidly by a one-way pipeline algorithm. Other examples of one-way pipeline algorithms include the carry pipelining for digit adders (see e.g., [Hallin and Flynn 72]) and pipeline arithmetic units (see e.g., [Ramamoorthy and Li 77]).

4.2. Two-way Pipeline Algorithms

There are inherent reasons why some problems can only be solved by pipeline algorithms using two-way data flows. We illustrate these reasons by examining three problems: band matrix-vector multiplication, recurrence evaluation, and priority queues.

Band Matrix-vector Multiplication

The band matrix-vector multiplication, for example, in Figure 7 differs from that in Figure 5 (a) in that the band in the matrix, the vector x , and the vector y can all be arbitrarily long. Thus, to solve the problem on a finite number of processors, all three quantities must move during the computation. This leads to the algorithm in Figure 8 (a), which uses the inner product step processor in Figure 8 (b). The x_i 's and y_i 's march in opposite directions, so that each x_i meets all the y_i 's. Notice that the x_i 's are separated by two time units, as are the y_i 's and the diagonal elements in the matrix. One can easily check that each y_p , initialized as zero, is able to accumulate all of its terms before it leaves the left-most processor.

A simple conclusion we can draw from this example is that if the size of the input and the output of a problem are larger than the size of the network, then all the inputs and intermediate results have to move during the computation. In this case, to achieve the greatest possible number of interactions among data we should let data flow in both directions simultaneously.

$$\begin{bmatrix}
 a_{11} & a_{12} & & & \\
 a_{21} & a_{22} & a_{23} & & \\
 a_{31} & a_{32} & a_{33} & a_{34} & \\
 a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\
 a_{51} & & & & \\
 & \ddots & & & \\
 & & 0 & &
 \end{bmatrix}
 \begin{bmatrix}
 x_1 \\
 x_2 \\
 x_3 \\
 x_4 \\
 \vdots \\
 \vdots
 \end{bmatrix}
 =
 \begin{bmatrix}
 y_1 \\
 y_2 \\
 y_3 \\
 y_4 \\
 \vdots \\
 \vdots
 \end{bmatrix}$$

Figure 7: Band matrix-vector multiplication.

In reference to Figure 8 (b), since a two-way pipeline algorithm makes each x_i meet all the y_j 's, it can compute the Cartesian product of the vectors x and y in parallel on a linear array. In this case, the a_{ij} , initialized as zero, is output from the bottom of the corresponding processor with a value resulting from some combination of x_i and y_j .

Matrix multiplication (or band matrix-vector multiplication) is of interest in its own right. Moreover many important computations such as convolution, discrete Fourier transform and finite impulse response filter are special instances of matrix-vector multiplications, and hence can be solved in parallel on linear processor arrays. For details, see [Kung and Leiserson 78].

Recurrence Evaluation

Many computational tasks are concerned with evaluations of recurrences. A k -th order recurrence problem is defined as follows: Given $x_0, x_{-1}, \dots, x_{-k+1}$, we want to evaluate x_1, x_2, \dots , defined by

$$x_i = R_i(x_{i-1}, \dots, x_{i-k}) \text{ for } i > 0, \quad (1)$$

where the R_i 's are given "recurrence functions". Parallel evaluation of recurrences is

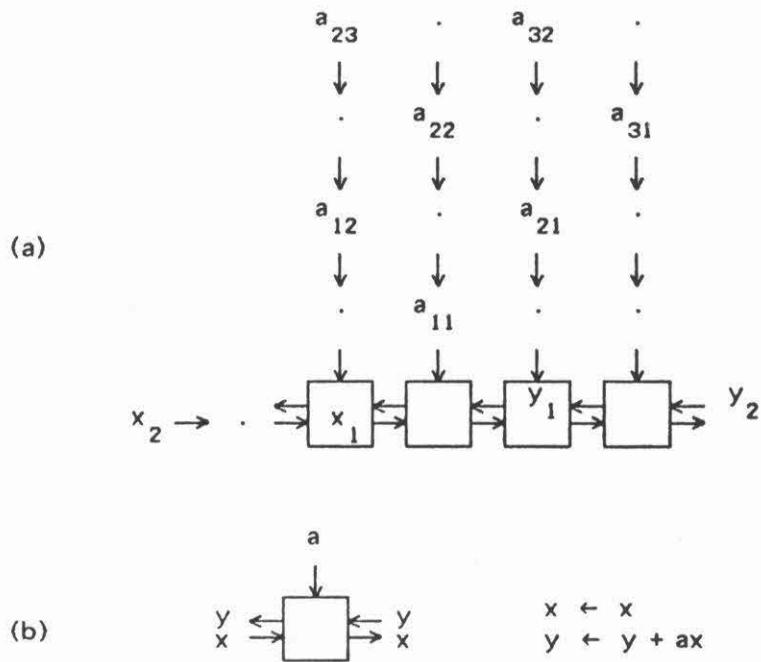


Figure 8: (a) A two-way pipeline computation for the band matrix-vector multiplication in Figure 7, and (b) the inner product step processor used.

interesting and challenging, since the recurrence problem on the surface appears to be highly sequential. We show that for a large class of recurrence functions, a k -th order recurrence problem can be solved in real-time on k linearly connected processors. That is, a new x_i is output every constant period of time, independent of k . To illustrate the idea, we consider the following linear recurrence:

$$x_i = ax_{i-1} + bx_{i-2} + cx_{i-3} + d, \quad (2)$$

where the a , b , c , and d are constants. It is easy to see that feedback links are needed for evaluating such a recurrence on a linear array, since every newly computed term has to be used later for computing other terms. A straightforward network with feedback loops for evaluating the recurrence is depicted in Figure 9, where each processor, except the right-most one which has more than one output port, is the inner product step processor of Figure 6. The x_i , initialized as d , gets cx_{i-3} , bx_{i-2} , and ax_{i-1} at time 1, 2, and 3, respectively.

At time 4, the final value of x_i is output from the right-most processor, and is also fed back to all the processors for use in computing x_{i+1} , x_{i+2} , and x_{i+3} .

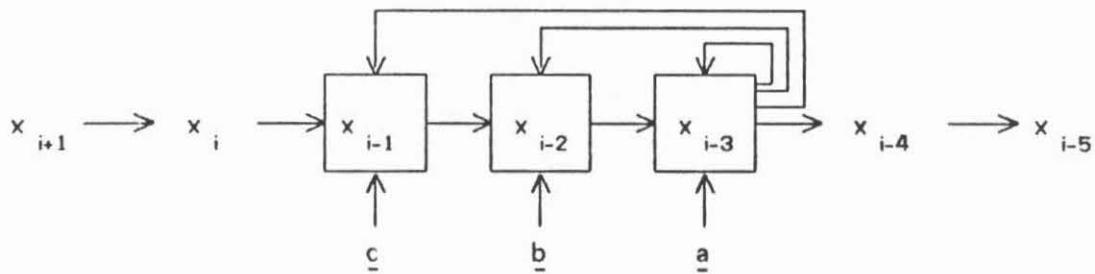


Figure 9: A linear array with feedback loops for evaluating the linear recurrence in Eq. (2).

The feedback loops in Figure 9 are undesirable, since they make the network irregular and non-modular. Fortunately, these feedback loops can be replaced with regular, two-way data flow. Assume that each processor is capable of performing the inner product step and also passing data, as depicted in Figure 10 (b). A two-way pipeline algorithm for evaluating the linear recurrence in Eq. (2) is schematized in Figure 10 (a). The additional processor, drawn in dotted lines, passes data only and is essentially a delay. Each x_i enters the right most processor with value zero, accumulates its terms as marching to the left, and feeds back its final value to the array through the left-most processor for use in computing x_{i+1} , x_{i+2} , and x_{i+3} . The final values of the x_i 's are output from the right-most processor at the rate of one output every two units of time.

The two-way pipeline algorithm for evaluating the linear recurrence described above extends directly to algorithms for evaluating any recurrences of the form:

$$x_i = F_1\{a_{i-1}, x_{i-1}, F_2[b_{i-2}, x_{i-2}, F_3(c_{i-3}, x_{i-3}, d_{i-4})]\}, \quad (3)$$

where the F_i 's are the functions and the a_i 's, b_i 's, c_i 's, d_i 's are the parameters which define the i -th recurrence function R_i (cf. Eq. (1)). Each x_i enters the right-most processor with the value d_{i-4} . The two-way pipeline algorithm for evaluating such a general recurrence is depicted in Figure 11 (a), using the generalized inner product step processor shown in Figure 11 (b). Recurrences of the form Eq. (3) include all linear recurrences and nonlinear

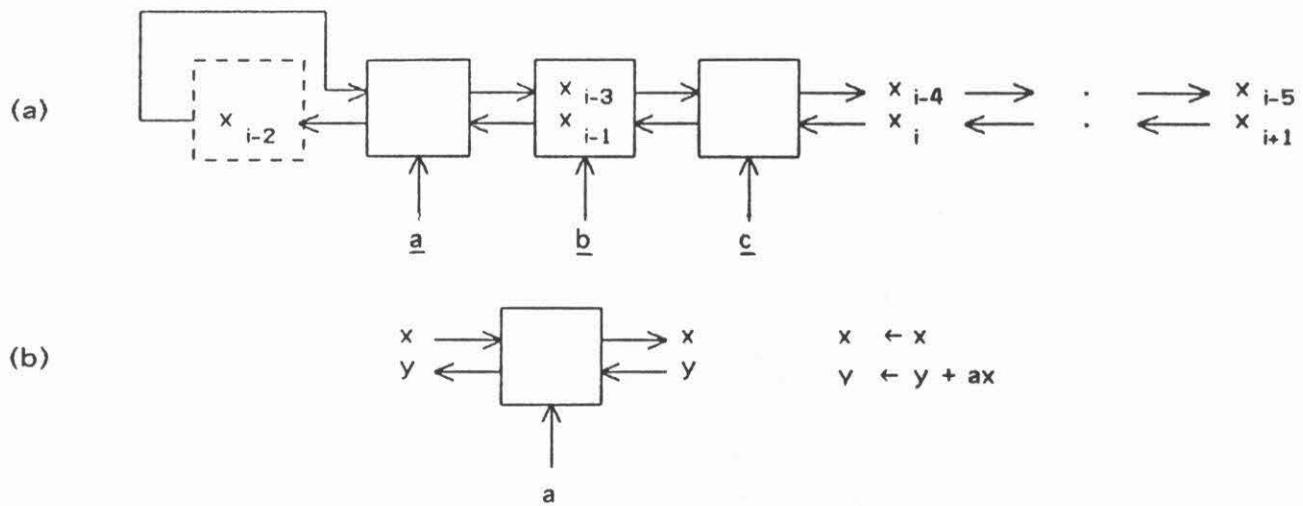


Figure 10: (a) A two-way pipeline algorithm for evaluating the linear recurrence in Eq. (2), and (b) the inner product step processor.

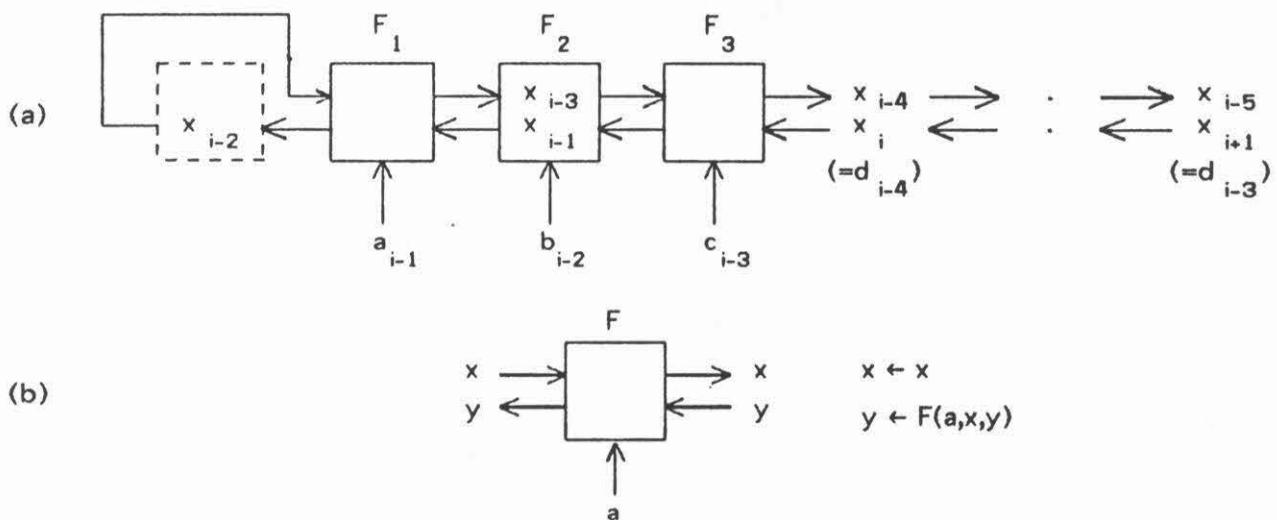


Figure 11: (a) A two-way pipeline algorithm for evaluating the general recurrence in Eq. (3), and (b) the generalized inner product step processor.

ones such as

$$x_i = 3x_{i-1}^2 + x_{i-2} * \sin(x_{i-3} + 4). \quad (4)$$

Eq. (4) corresponds to the case where $F_3(x, y, z) = \sin(y + z)$ with $z=4$, $F_2(x, y, z) = y * z$, and $F_1(x, y, z) = 3y^2 + z$. In fact, Eq. (3) is not yet the most general form of recurrence that linear processor arrays can evaluate in real-time. For example, the generalized inner product step processor in Figure 11 (b) can be further generalized to include the capability of updating both x and y . That is, the processor performs $x \leftarrow F^{(1)}(a, x, y)$ and $y \leftarrow F^{(2)}(a, x, y)$ according to some given functions $F^{(1)}, F^{(2)}$. Given a linear array of such generalized inner product step processors, it is often an interesting and nontrivial task to figure out what recurrence the array actually evaluates. Here we note without proof that the problem can always be solved in principle at least by using induction on the number of processors in the array.

We conclude our discussion of recurrence evaluation by stating that two-way pipelining is a powerful construct in the sense that it can eliminate the need for using undesirable feedback loops such as those encounter in Figure 9.

Priority Queues

A data structure that can process `INSERT`, `DELETE`, and `EXTRACT_MIN` operations is called a priority queue. Priority queues are basic structures used in many programming tasks. If a priority queue is implemented by some balanced tree, for example a 2-3 tree, then an operation on the queue will typically take $O(\log n)$ time when there are n elements stored in the tree [Aho et al. 75]. This $O(\log n)$ delay can be replaced with a constant delay if a linear array of processors is used to implement the priority queue. Here we shall only sketch the basic idea behind the linear array implementation. A complete description will be reported in another paper.

To visualize the algorithm, we assume that the linear array in Figure 4 has been physically rotated 90° and that processors are capable of performing comparison-exchange operations on elements in neighboring processors. We try to maintain elements in the array in sorted order according to their weights. After an element is inserted into the array from the top, it will "sink down" to the proper place by trading positions with elements having smaller weights (so lighter elements will "bubble up"). To delete an element, we insert an "anti-element" which first sinks down from the top to find the element, then annihilates it. Elements below can then bubble up into the empty processor. Hence the element with the smallest weight will always be kept at the top of the processor,

and is ready to be extracted in constant time. An important observation is that "sinking down" or "bubbling up" operations can be carried out concurrently at various processors throughout the array. For example, the second insertion can start right after the first insertion has passed the top processor. In this way, any sequence of n INSERT, DELETE, or EXTRACT_MIN operations can be done in $O(n)$ time on a linear array of n processors, rather than $O(n \log n)$ time as required by a uniprocessor. In particular, by performing n INSERT operations followed by n EXTRACT_MIN operations the array can sort n elements in $O(n)$ time, where the sorting time is completely overlapped with input and output. A similar result on sorting was recently proposed by [Chen et al. 78]. They do not, however, consider the deletion operation.

5. Algorithms Using Two Dimensional Arrays

5.1. Algorithms Using Square Arrays

The square array, as shown in Figure 12, is perhaps one of the first communication structures studied by researchers who were interested in parallel processing.

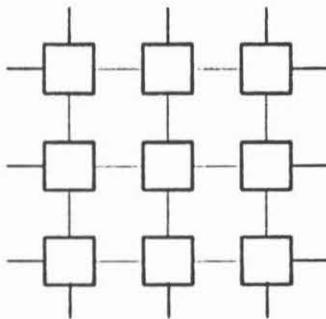


Figure 12: A 3x3 square array.

Work in cellular automata, which is concerned with computations distributed in a two-dimensional orthogonally connected array, was initiated by [Von Neumann 66]. From an algorithmic point of view, the square array structure is natural for problems involving matrices. These problems include graph problems defined in terms of adjacency matrices, and numerical solutions to discretized partial differential equations. Cellular algorithms for pattern recognition have been proposed in [Kosaraju 75, Smith 71], for graph problems in

[Levitt and Kautz 72], for switching in [Kautz et al. 68], for sorting in [Thompson and Kung 77], and for dynamic programming in [Guibas et al. 79]. The algorithms for dynamic programming in [Guibas et al. 79] are quite special in that they involve data being transmitted at two different speeds, which give the effect of "time reverse" for the order of certain results. For numerical problems, much of the research on the use of the square structure is motivated or influenced by the ILLIAC IV computer, which has an 8x8 processor array. The broadcast capability provided by the ILLIAC IV is useful in communicating relaxation and termination parameters required by many numerical methods. This suggests that for VLSI implementation some additional broadcast facility be provided on the top of the existing square array connection. This, however, would certainly complicate the chip layout.

5.2. Algorithms Using Hexagonal Arrays

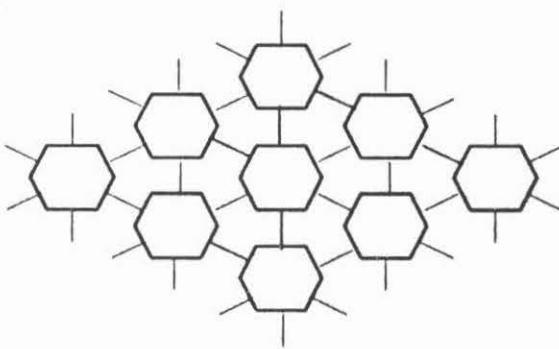


Figure 13: A 3x3 hexagonal array

The hexagonal array structure, as shown in Figure 13, enjoys the property of symmetry in three directions. Therefore, after a binary operation is executed at a processor, the result and two inputs can all be sent to the neighboring processor in a completely symmetric way. A good example is the matrix multiplication algorithm considered in Section 2, where elements in matrices A, B, and C all circulate throughout the network (cf. Figure 3). This type of computation eliminates a possible separate loading or unloading phase, which is typically needed in algorithms using square array structures.

We know of two other problems that can be solved naturally on hexagonal arrays: LU

decomposition [Kung and Leiserson 78] and transitive closure [Guibas et al. 79]. We indicate below that, in some sense, these two problems and the matrix multiplication problem are all defined by recurrences of the "same" type. Thus, it is not coincidental that they can be solved by similar algorithms using hexagonal arrays. The defining recurrences for these problems are as follows:

Matrix Multiplication

$$\begin{aligned}
 (*) \quad c_{ij}^{(1)} &= 0, \\
 c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik} b_{kj}, \\
 c_{ij} &= c_{ij}^{(n+1)}.
 \end{aligned}$$

LU-decomposition

$$\begin{aligned}
 (*) \quad a_{ij}^{(1)} &= a_{ij}, \\
 a_{ij}^{(k+1)} &= a_{ij}^{(k)} + l_{ik}(-u_{kj}), \\
 l_{ik} &= \begin{cases} 0 & \text{if } i < k, \\ 1 & \text{if } i = k, \\ a_{ik}^{(k)} u_{kk}^{-1} & \text{if } i > k, \end{cases} \\
 u_{kj} &= \begin{cases} 0 & \text{if } k > j, \\ a_{kj}^{(k)} & \text{if } k \leq j. \end{cases}
 \end{aligned}$$

Transitive Closure

$$\begin{aligned}
 (*) \quad a_{ij}^{(1)} &= a_{ij}, \\
 a_{ij}^{(k+1)} &= a_{ij}^{(k)} + a_{ik}^{(k)} a_{kj}^{(k)}.
 \end{aligned}$$

Notice that the main recurrences, denoted by (*), of the three problems have similar structures for subscripts and superscripts.

6. Algorithms Using Trees

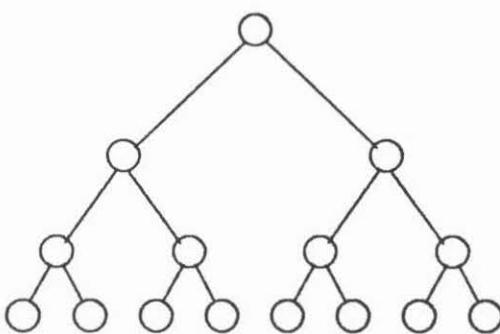


Figure 14: The tree structure.

6.1. Characteristics of the Tree Structure

The tree structure, shown in Figure 14, supports logarithmic-time broadcast, search, or fan-in, which is theoretically optimal. The root is the natural I/O node for outside world communication. In this case, a small problem can be solved on the top portion of a large tree. Hence a tree structure in principle can support problems of any size that can be accommodated, without performance penalty. Figure 15 shows an interesting "H" shaped layout of a binary tree, which is convenient for placement on a chip [Mead and Rem 78].

6.2. Tree Algorithms

The logarithmic-time property for broadcasting, searching, and fan-in is the main advantage provided by the tree structure that is not shared by any array structure. The tree structure, however, has the following possible drawback. Processors at high levels of the tree may become bottlenecks if the majority of communications are not confined to processors at low levels. We are interested in algorithms that can take advantage of the power provided by the tree structure while avoiding this drawback of the structure.

Search Algorithms

The tree structure is ideal for searching. Assume, for example, that information stored at

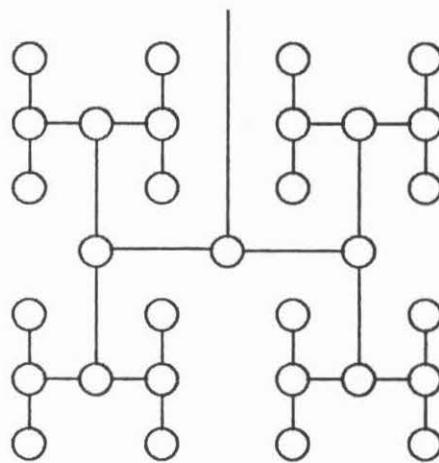


Figure 15: Embedding a binary tree in a two-dimensional grid.

the leaves of a tree forms the data base. Then we can answer questions of the following kinds rapidly: "What is the nearest neighbor of a given element?", "What is the rank of a given element?", "Is a given element inside a certain subset of the data base?" The paradigm to process these queries consists of three phases: (i) the given element is broadcast from the root to leaves, (ii) the element is compared to some relevant data at every leaf simultaneously, and (iii) the comparison results from all the leaves are combined into a single answer at the root, through some fan-in process. It should be clear that using the paradigm and assuming appropriate capabilities of the processors, queries like the ones above can all be answered in logarithmic time. Furthermore, we note that when there are many queries, it is possible to pipeline them on the tree.

A similar idea has been pointed out in [Browning 79]. Algorithms which first generate a large number of solution candidates and then select from among them the true solutions can be efficiently supported by the tree structure. NP-complete problems [Karp 72] such as

the clique problem and the color cost problem are solvable by such algorithms. One should note that with this approach an exponential number of processors will be needed to solve an NP-complete problem in polynomial time. However, with the emergence of VLSI this brute force approach may gain importance. Here we merely wish to point out that the tree structure matches the structure of some algorithms that solve NP-complete problems.

Systolic Search Tree

As one is thinking about applications using trees, data structures such as search trees (see, for example, [Aho et al. 75, Knuth 73]) will certainly come to mind. The problem is how to embed a balanced search tree in a network of processors connected by a tree so that the $O(\log n)$ performance for the INSERT, DELETE, and FIND operations can be maintained. The problem is nontrivial because most balancing schemes require moving pointers around, but the movement of pointers is impossible in a physical tree where pointers are fixed wires. To get the effect of balancing in the physical tree, data rather than pointers must be moved around. Common balanced tree schemes such as AVL trees and 2-3 trees do not map well onto the tree network because data movements involved in balancing are highly non-local. A new organization of a hardware search tree, called a systolic search tree, was recently proposed by [Leiserson 79], on which the data movements for balancing are always local so that the requirement of $O(\log n)$ performance can be satisfied. In [Leiserson 79], an application of using the systolic search tree as a common storage for a collection of disjoint priority queues is discussed.

Evaluation of Arithmetic Expressions and Recurrences

Another application of the tree structure is its use for evaluating arithmetic expressions. Any expression of n variables can be evaluated by a tree of at most $4\lceil \log_2 n \rceil$ levels [Brent 74], but the time to input the n variables to the tree from the root is still $O(n)$. This input time can often be overlapped with the computation time in the case of evaluating recurrences. The idea of two-way pipeline algorithms for evaluating recurrences on linear arrays (cf. Figure 11 (a)) extends directly to trees. Corresponding to the inner product step processor in Figure 11 (b), for a tree we now have processors of the form shown in

Figure 16, which are defined in terms of some given functions F , G_1 , and G_2 .

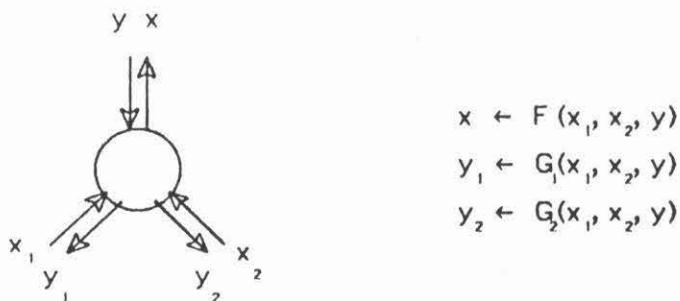


Figure 16: The generalized inner product step processor for trees.

The tree structure can be used to evaluate systems of recurrences. The final values of the components of each term (which is a vector) are available at leaf processors, and are fed back to the tree from the leaves for use in computing other terms. It is instructive to note that all of the tree algorithms mentioned above correspond to various definitions of the functions F , G_1 , and G_2 at each processor (cf. Figure 16.)

7. Algorithms Using Shuffle-Exchange Networks

Consider a network having $n=2^m$ nodes, where m is an integer. Assume that nodes are named $0, 1, \dots, 2^m-1$. Let $i_m i_{m-1} \dots i_1$ denote the binary representation of any integer i , $0 \leq i \leq 2^m-1$. The shuffle function is defined by

$$S(i_m i_{m-1} \dots i_1) = i_{m-1} i_{m-2} \dots i_1 i_m$$

and the exchange function is defined by

$$E(i_m i_{m-1} \dots i_1) = i_m \bar{i}_{m-1} \dots \bar{i}_2 \bar{i}_1.$$

The network is called a shuffle-exchange network if node i is connected to node $S(i)$ for all i , and to node $E(i)$ for all even i . Figure 17 is a shuffle-exchange network of size $n=8$.

Observe that by using the exchange and shuffle connections alternately, data at pairs of nodes whose names differ by 2^i can be brought together for all $i = 0, 1, \dots, m-1$. This type of communication structure is common to a number of algorithms. It is shown in [Batcher 68] that the bitonic sort of n elements could be carried out in $O(\log^2 n)$ steps on the shuffle-exchange network when the processing elements are capable of performing comparison-exchange operations. It is shown in [Pease 68] that the n -point fast Fourier

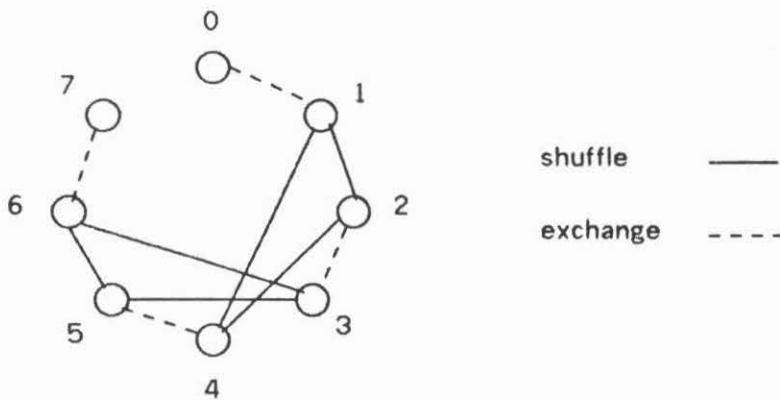


Figure 17: A shuffle-exchange network.

transform could be done in $O(\log n)$ steps on the network when the processing elements are capable of doing addition and multiplication operations. Other applications including matrix transposition and linear recurrence evaluation are given in [Stone 71, Stone 75]. The two articles by Stone give clear expositions for all these algorithms and have good discussions on the basic idea behind them.

Many powerful rearrangeable permutation networks, such as those in [Benes 65] which are capable of performing all possible permutations in $O(\log n)$ delays, can be viewed as multi-stage shuffle-exchange networks (see, e.g., [Kuck 78]). The shuffle-exchange network, perhaps due to its great power in permutation, suffers from the fact that its structure has a very low degree of regularity and modularity. This can be a serious drawback, as far as VLSI implementations are concerned. Indeed, it was recently shown by [Thompson 79] that the network is not planar and cannot be embedded in silicon using area linearly proportional to the number of nodes.

8. Concluding Remarks

Many problems can be solved by algorithms that are "good" for VLSI implementation. The communication geometries based on the array and tree structure or their combinations seem to be sufficient for solving a large class of problems. When a large problem is to be solved on a small network, one can either decompose the problem or decompose an algorithm that requires a large network [Kung 79].

Algorithms employing multi-directional data flow can realize extremely complex computations, without violating the simplicity and regularity constraints. Moreover, these algorithms do not require separate loading or unloading phases. We believe that hexagonal connection is fundamentally superior to square connection, because the former supports data flows in more directions than the latter and the two structures are about of the same complexity as far as implementations are concerned.

We need a new methodology for coping with the following problems:

- Notation for specifying geometry and data movements.
- Correctness of algorithms defined on networks.
- Guidelines for design of VLSI algorithms.

It is seen in this paper that there is a close relationship between the defining recurrence of a problem and the VLSI algorithms for solving the problem. This association deserves further research. We hope that eventually the derivation of good VLSI algorithms based on given recurrences will be largely mechanical. An initial step towards this goal has been independently taken by D. Cohen [Cohen 78].

ACKNOWLEDGMENTS

Comments by R. Hon, P. Lehman, S. Song, J. Bentley, C. Thompson and M. Foster at CMU are appreciated.

References

- [Aho et al. 75] Aho, A., Hopcroft, J.E. and Ullman, J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1975.
- [Batcher 68] Batcher, K.E.
Sorting networks and their applications.
1968 Spring Joint Computer Conf. 32:307-314, 1968.
- [Benes 65] Benes, V.E.
Mathematical Theory of Connecting Networks and Telephone Traffic.
Academic Press, New York, 1965.
- [Brent 74] Brént, R.P.
The Parallel Evaluation of General Arithmetic Expressions.
Journal of the ACM 21(2):201-206, April 1974.

- [Browning 79] Browning, S.
Algorithms for the Tree Machine.
To appear in the forthcoming book, *Introduction to VLSI Systems*, by C. A. Mead and L. A. Conway, Addison-Wesley.
- [Chen 75] Chen, T.C.
Overlap and Pipeline Processing, pages 375-431.
In *Introduction to Computer Architecture*, (Stone, H.S., Editor), Science Research Associates, 1975.
- [Chen et al. 78] Chen, T.C., Lum, V.Y. and Tung, C.
The Rebound Sorter: An Efficient Sort Engine for Large Files
Proceedings of the 4th International Conference on Very Large Data Bases, IEEE, pages 312-318, 1978.
- [Cohen 78] Cohen, D.
Mathematical Approach to Computational Networks.
Technical Report ISI/RR-78-73, University of Southern California, Information Sciences Institute, November 1978.
- [Guibas et al. 79] Guibas, L.J., Kung, H.T. and Thompson, C.D.
Direct VLSI Implementation of Combinatorial Algorithms
Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, January, 1979.
- [Hallin and Flynn 72] Hallin, T.G. and Flynn, M.J.
Pipelining of Arithmetic Functions.
IEEE Trans. on Comp. C-21:880-886, 1972.
- [Karp 72] Karp, R. M.
Reducibility Among Combinational Problems, pages 85-104.
In *Complexity of Computer Computations*, Plenum Press, New York, 1972.
- [Kautz et al. 68] Kautz, W.H., Levitt, K.N. and Waksman, A.
Cellular Interconnection Arrays.
IEEE Transactions on Computers C-17(5):443-451, May 1968.
- [Knuth 73] Knuth, D. E.
The Art of Computer Programming. Volume 3: Sorting and Searching. Addison-Wesley, 1973.
- [Kosaraju 75] Kosaraju, S.R.
Speed of Recognition of Context-Free Languages by Array Automata.
SIAM J. on Computing 4:331-340, 1975.
- [Kuck 78] Kuck, D. J.
The Structure of Computers and Computations. John Wiley and Sons, New York, 1978.
- [Kung 79] Kung, H. T.

- The Structure of Parallel Algorithms.*
 In *Advances in Computers*, (Yovits, M. C., Editor), Academic Press, New York, 1979.
- [Kung and Leiserson 78]
 Kung, H. T. and Leiserson, C. E.
Systolic Arrays (for VLSI).
 Technical Report, Carnegie-Mellon University, Department of Computer Science, December 1978.
 To appear in the forthcoming book, *Introduction to VLSI Systems*, by C. A. Mead and L. A. Conway, Addison-Wesley, 1979.
- [Leiserson 79] Leiserson, C. E.
Systolic Priority Queues
 Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication, California Institute of Technology, January, 1979.
- [Levitt and Kautz 72]
 Levitt, K.N. and Kautz, W.H.
 Cellular Arrays for the Solution of Graph Problems.
Communications of the ACM 15(9):789-801, September 1972.
- [Mead and Conway 79]
 Mead, C. A. and Conway, L. A.
Introduction to VLSI Systems.
 Addison-Wesley, 1979.
- [Mead and Rem 78]
 Mead, C. and Rem, M.
Cost and Performance of VLSI Computing Structures.
 Technical Report 1584, California Institute of Technology, Department of Computer Science, 1978.
- [Pease 68] Pease, M.C.
 An Adaptation of the Fast Fourier Transform for Parallel Processing.
Journal of the ACM 15:252-264, April 1968.
- [Ramamoorthy and Li 77]
 Ramamoorthy, C.V. and Li, H.F.
 Pipeline Architecture.
Computing Surveys 9(1):61-102, March 1977.
- [Smith 71] Smith III, A.R.
Two-Dimensional Formal Languages and Pattern Recognition by Cellular Automata
 12th IEEE Symposium on Switching and Automata Theory, pages 144-152, 1971.
- [Stone 71] Stone, H.S.
 Parallel Processing with the Perfect Shuffle.
IEEE Transactions on Computers C-20:153-161, February 1971.

[Stone 75]

Stone, H.S.

Parallel Computation, pages 318-374.

In *Introduction to Computer Architecture*, (Stone, H.S., Editor), Science Research Associate, Chicago, 1975.

[Sutherland and Mead 77]

Sutherland, I. E. and Mead, C. A.

Microelectronics and Computer Science.

Scientific American 237:210-228, 1977.

[Thompson and Kung 77]

Thompson, C.D. and Kung, H.T.

Sorting on a Mesh-Connected Parallel Computer.

Communications of the ACM 20(4):263-271, April 1977.

[Thompson 79]

Thompson, C.D.

Area-Time Complexity for VLSI

Eleventh Annual ACM Symposium on Theory of Computing, May, 1979.

[Von Neumann 66] Von Neumann, J.

Theory of Self-Reproducing Automata.

(Burks, A. W., Editor), University of Illinois Press, Urbana, Illinois, 1966.

FABRICATION SESSION

Chairperson:

Lynn A. Conway, Xerox Corporation, and Visiting Associate Professor of Electrical Engineering and Computer Science at M.I.T.

This session concerns the fabrication of very large scale integrated systems. For the purposes of the session, we will consider fabrication to include all procedures for processing both information and artifacts during the implementation of integrated systems. Let's think of fabrication as the transformation of a chip's design file or "mask spec" into packaged chips ready for functional testing.

As we reflect on the history of electronics, most of us visualize the steady improvements in fabrication technology, and the resulting steady increases in the density of integrated circuitry, as the basic driving force behind the past integrated circuit and present integrated system revolutions. We can also look ahead and clearly see that these increases in density will continue for many years into the future, before fundamental physical limits block further progress. The means for achieving these density increases are known. The effects of scaling down the physical dimensions of devices and circuits are also known. We can safely predict several orders of magnitude increase in density before really fundamental limits are reached in semiconductor technology.

Well, this is a situation something like inflation: we've been used to it for a long time, and expect it to continue. We begin to plan for it, and to discount it when looking ahead in our other integrated system activities in architecture and design.

However, is that all there is to the improvement of fabrication, just making denser, faster circuitry as time goes by? Aren't there some other important dimensions to be explored, other areas for innovation in integrated system fabrication?

Five papers will be presented in this session. Each explores some dimension for the improvement of fabrication other than simply increasing circuit density and speed.

The first two papers, though concerning rather different technologies and design regimes, both discuss fabrication ideas in a systems context: ideas which might simplify and speed up system design and testing, or improve our capability for effectively coping with fabrication defects. The paper by J. Raffel proposes the use of a form of restructurable logic as a method for utilizing higher density and larger chip area, even in the presence of defects. The paper describes a technique for fabricating systems containing non-volatile programmable links as a device type, and techniques for designing, testing, and restructuring VLSI systems implemented with this fabrication technology. The paper by H. Muller, H. Stopper, and R. Tam describes fabrication related techniques for on-chip regulation, test/diagnostic monitors, and signature circuits, and the application of these techniques in a high speed LSI chip family for main-frame design.

The following two papers describe specific research results in fabrication which might have useful system applications or provide improved data for planning fabrication facilities. Hierarchical system design would be further enabled if more levels of interconnect were available. The two layer metal process described by R. Huber provides one such additional level. It is also a very simple process which shows promise in enabling the reduction of fabrication times. The use of high density electron beams for rapid exposure of VLSI patterns may play an important role in VLSI system prototyping and manufacturing. The paper by T. Sasaki provides the quantitative results of a study of an important limiting effect in the application of such high density beams.

The final paper, by R. Hon, explores a different dimension of fabrication: how to optimize the entire implementation sequence to best support the prototyping of complex VLSI designs. This paper asks some key questions: how long does it take to go from a design file to a packaged chip, where is the time consumed, how can the time be reduced, and how might the designer's interface to implementation be simplified? These are rather direct and simple questions, but how many of us know the answers for our own particular organization? The answers may be extremely important in a VLSI system prototyping environment. Present day chip design efforts usually involve large teams of specialists, and time consuming, conservative design strategies, leading to high design costs and long lead times. These costly and conservative approaches may be largely due to the very long implementation turnaround times common in the industry. The reduction of implementation times and simplification of implementation procedures, as proposed by Hon, would enable students of integrated system design to "learn by doing", and designers of large complex VLSI systems to build modularly and hierarchically, testing as they proceed, as is now done in the software world when building complex systems.

Following the presentations of the papers, the group of speakers will be joined by Merrill Brooksby, Corporate Design Aids Program Manager, Hewlett-Packard Corporation, and Tom Griswald, Supervisor of the LSI Technology Group at Caltech JPL, for a panel discussion concerning possible future developments in VLSI fabrication.

On the Use of Nonvolatile Programmable Links
for Restructurable VLSI^{*}

J.I. Raffel

MIT Lincoln Laboratory, Lexington, Massachusetts

VLSI - Objectives, Problems and History

There seems to be general agreement that VLSI implies the fabrication of digital logic circuits having minimum feature definition at least as small as one micron and levels of integration of greater than one hundred thousand gates per package.

Associated with VLSI are processing and device performance limitations which have been identified and extensively studied as well as a variety of architectural and digital design strategies which have received only cursory treatment. Much attention has been given to the use of x-ray, electron-beam and other lithographic techniques for achieving submicron device geometries. Ion implantation and the scaling laws for minimizing short channel effects in FET's have been widely discussed, but relatively little has been proposed in the way of solutions to the yield problems raised by the combination of very small devices and such high levels of integration.

Specifically, past improvements in circuit integration have almost universally been predicated on brute force improvements in lowering defect density to achieve economically viable yields of perfect devices. There are reasons to believe that with VLSI this is neither a possible nor necessarily desirable objective. The structures we propose to fabricate are now so complex and catastrophic defect size so small that, on the one hand, there is a great likelihood that we can never make them perfect and, on the other, the possibility that, for the first time, we can afford the degree of redundancy necessary to provide fault-tolerant operation on a significant scale. The effective use of redundancy to achieve acceptable yield levels introduces difficult problems in device implementation, layout, logic-design and testing.

We choose at the outset to define as an appropriate goal not simply the utilization of higher gate density to proceed from LSI to VLSI but the

* This work was sponsored by the Department of the Air Force.

The views and conclusions contained in this document are those of the contractor and should not be interpreted as necessarily representing the official policies either expressed or implied, of the United States Government.

simultaneous expansion of chip size to whole wafer technology to provide a much greater increase in overall chip complexity than could be achieved by gate shrinkage alone. The decision to compound these individual contributions to chip capacity is based on the belief that once a level of complexity is reached which requires restructurability, old thresholds are erased and the tradeoffs between chip size and yield are fundamentally altered.

Restructurable Logic

The ability to reconfigure or restructure the logic of a monolithic integrated circuit may be used to accomplish three different sets of objectives which are sometimes confused.

It is proposed that defect avoidance, the first of these, is probably essential for the levels of integration and resolution required for VLSI. This category may be further subdivided depending on whether restructuring is accomplished only at wafer probe or whether faults which develop in the field are also correctible.

The ability to restructure for functional specialization is a way of achieving customized structures with standard modules thus reducing the long turn-around times due to design, layout, fabrication, and testing associated with all-custom designs.

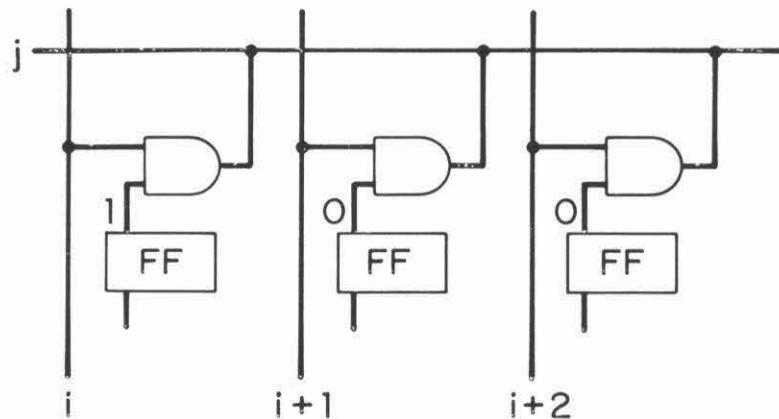
A third category utilizes dynamic reconfiguration to alter the system configuration during the running of a problem to increase machine efficiency by providing better utilization of resources to meet the changing computational requirements of the application.

Each of these objectives presents different requirements relative to diagnostic techniques, interface requirements, speed and frequency of restructuring, implications of volatility and permissible overhead.

Programmable Links – Volatility vs. Nonvolatility

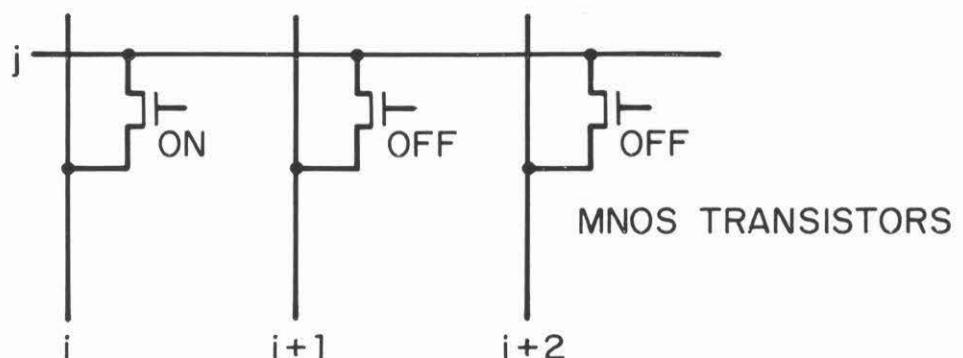
It is possible to excise faulty sections of logic and patch in healthy ones by switching path connectivity under the control of stored bit patterns which are alterable on the basis of test and diagnostic results. The approaches to this capability vary considerably depending on whether the control store is volatile or nonvolatile. In the former case testing and reconfiguration must be initiated each time power has been removed from the part in question. In general there is also significantly more area associated with volatile link control than with a nonvolatile structure, although the former has the advantage of using basic components which are identical to those used in the internal logic structure.

Figure 1 shows an example of a volatile programmable link and Fig. 2 shows an example of a non-volatile programmable link. For concreteness an MNOS structure¹ is assumed but nonvolatile operation has been achieved with other devices.^{2,3} In addition to the larger area required, the volatile FF link control requires significant standby power or, if a dynamic storage cell



VOLATILE PROGRAMMABLE LINKS

FIG. 1



NONVOLATILE PROGRAMMABLE LINKS

FIG. 2

is used, periodic refreshing.

The general scheme for using nonvolatile programmable links to restructure a VLSI circuit is shown in Fig. 3. A typical "cell" which may be of MSI to LSI complexity is connected to its environment by horizontal lines fabricated on second level metal and vertical lines on first. In order to connect any output node to any input node, it is only necessary to connect three links: one output link, one crossover link, and one input link, thereby forming a continuous path from output to input.

Figure 4 shows an array of such cells and busses forming a chip along with estimates of the numbers of gates, cells, busses and links for a VLSI complexity varying between 10^5 and 10^7 gates. These numbers are purely speculative and are only meant to help focus on some of the implementation problems and architectural implications of this approach. Although busses are shown schematically to run the full chip length, it is proposed that these would, in fact, be segmented, providing a mix of both local and express runs for greater flexibility in routing in much the same way as has been proposed for Programmed Logic Arrays.⁴

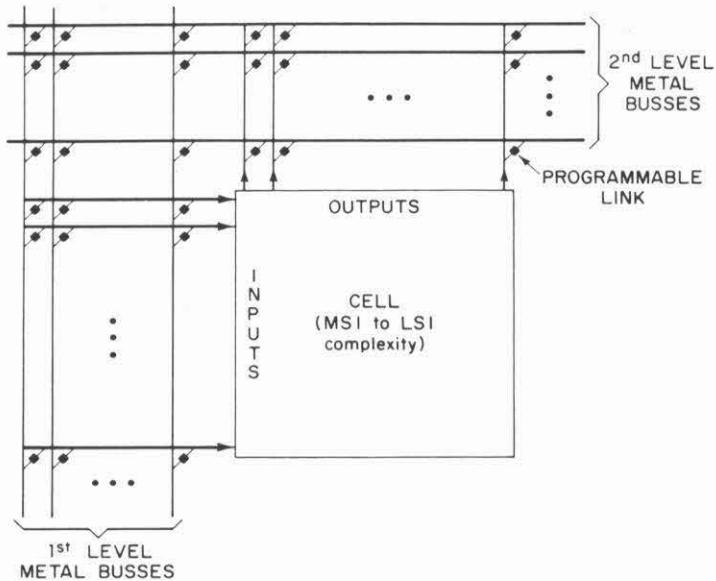
Nonvolatile Programmable Link Technology

The feasibility of programmable link control depends critically on the ability to access a sufficiently large number of links in order to provide a flexible reconfiguration capability from a practical number of extra programming nodes. A decoding tree structure must be developed that provides the necessary translation. Preliminary estimates indicate that the overhead associated with such a system may not be unreasonable. Estimates of area requirements for links themselves also seem to represent an acceptable percentage of total chip area, although there is a direct tradeoff between the link electrical conductivity and the area consumed, which requires detailed study. Figure 5 shows one proposed layout of a nonvolatile programmable link.

In order to program an NPL it is necessary to provide coincident voltages to the metal gate electrode and the orthogonal, isolated, n-silicon stripe whose intersection defines the MNOS transistor channel region. A scheme for fabricating high density, planar, isolated digit lines has been described previously.⁵ A chip having 1000 cells each with 10 inputs, 10 outputs, and 20 horizontal and 20 vertical busses, would require 200 input links, 200 output links and 400 crossover links. An array of 1000 gate lines and 1000 silicon lines would then select one out of the approximate total of one million links thus requiring only 20 bits for programming control.

Testing, Diagnosis, Routing and Rerouting

Depending on the degree of restructurability desired the hardware for testing, diagnosing, routing and rerouting will vary considerably. In the simplest case where only defect avoidance is desired, this equipment may be located off-chip. This of course enormously simplifies the system by totally bypassing the question of how one tests the tester. Even in this simplest



CELL, BUSSES AND PROGRAMMABLE LINKS

FIG. 3

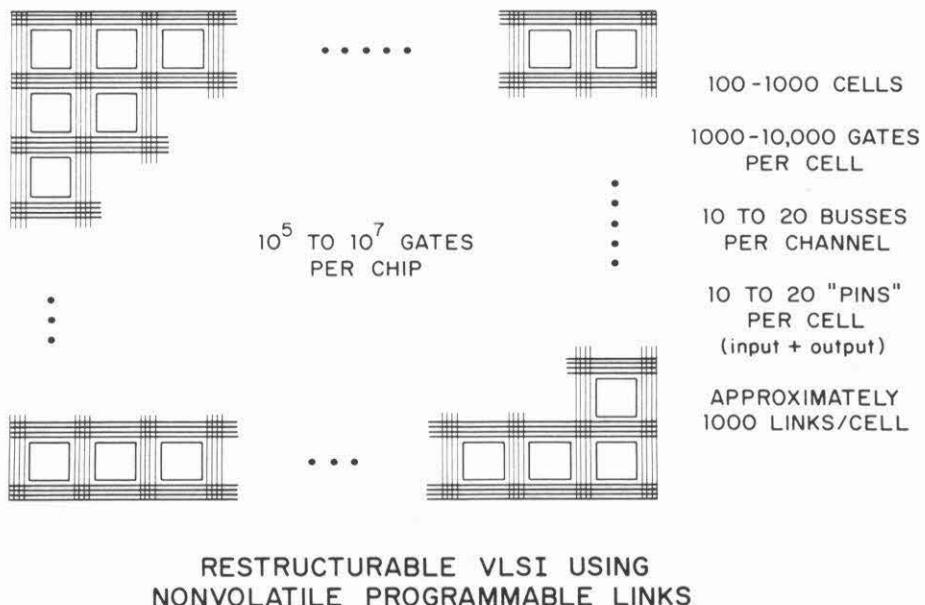


FIG. 4

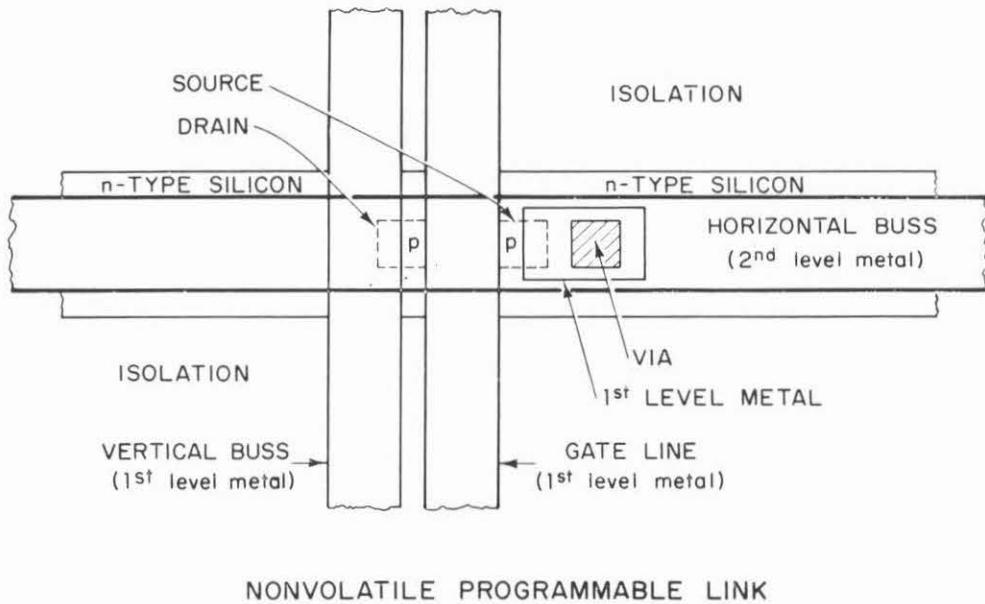
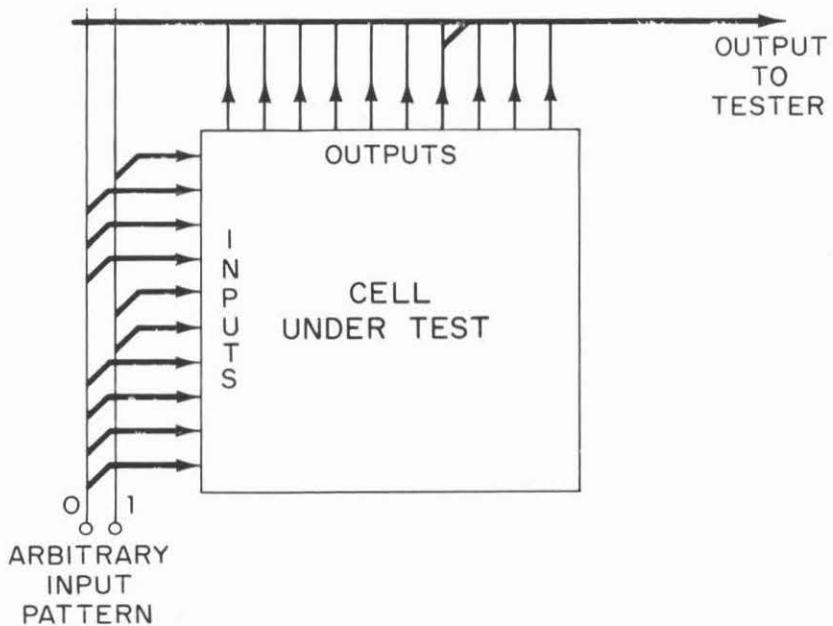


FIG. 5



TESTING A CELL WITH PROGRAMMABLE LINKS

FIG. 6

case the problems of accessing, testing and rerouting around failed areas will require extremely sophisticated computer design aids.

One powerful feature of NPL's is that it is possible to isolate and access individual cells from the chip exterior. This is illustrated in Fig. 6 where links to a single cell are activated with all other cells disconnected from the buss. An arbitrary pattern of ONES and ZEROS may be presented at the cell inputs while a single cell output is monitored. Thus each cell may be tested singly before full chip routing is initiated.

It may even be possible to allocate one region of the chip to implementation of a tester programmer (T-P). This on-chip circuit would be first tested, programmed and routed by use of an off-chip tester-programmer and, when fully configured, could then replace the external T-P for any testing or rerouting that occurs after system deployment. To insure the integrity of the on-chip T-P, classical error-correcting techniques may be employed.

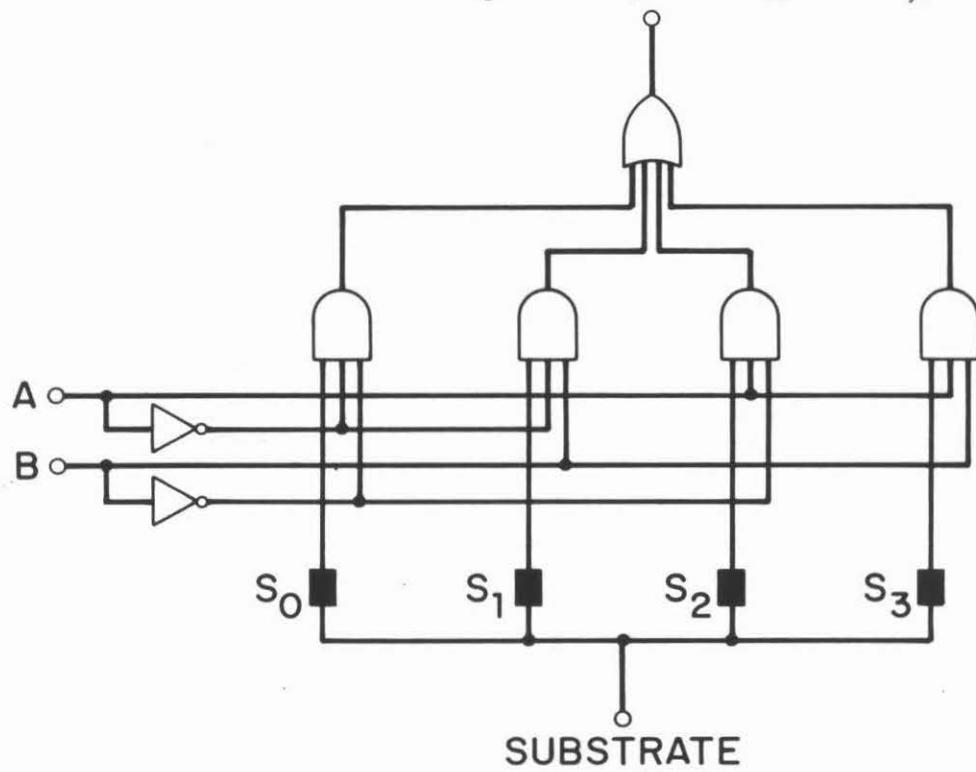
Cell Design, Customization

An understanding of the spectrum of options available in cell or module design is critical to effective restructurability. These range from uniform arrays of standard cells with local customization links to totally customized cell and interconnect designs. Between these extremes lie a mix of standard cells and an array of custom cells on a regular grid structure. Each of these represents a different point on the tradeoff curve between flexibility and control simplicity. A special case must be made in the instance of memory. While, in principle, it is possible to reduce both logic and memory to basic AND/OR gates, the high usage of storage and the potential for high packing-density (because of locally connected topology) warrants providing special cells for memory.

The use of nonvolatile links to perform cell customization, as distinct from interconnect routing for defect avoidance, is illustrated by the simple example of Fig. 7. Here, local customization links are used to determine the logic function of a universal sub-cell which might be part of the larger cell shown in Fig. 3. By activating appropriate combinations of customization links S_0 , S_1 , S_2 , and S_3 it is possible to produce the 16 truth tables for the 2 input variables A and B. Note that no extra busses are required since links are connected to the substrate.

One of the principal design parameters affecting restructurability is the degree of connectivity available for routing around failed sections. There has been considerable theoretical analysis of cellular logic with nearest neighbor connectivity, and simple buss-structured multiprocessor systems, etc. Unfortunately none of these provides much insight in treating highly connected random logic. There are basic engineering tradeoffs between module complexity and connectivity; the more complex the basic module unit the higher the gate-to-pin ratio and the smaller the control structure necessary to control all connection to its environment. Balancing this is the falloff in yield with module complexity. The use of intercell

$$\text{OUTPUT} = S_0 \bar{A} \bar{B} + S_1 \bar{A} B + S_2 A \bar{B} + S_3 A B$$



LOCAL CUSTOMIZATION LINKS

FIG. 7

multiplexing could significantly affect this design balance. These relationships between gate count, pin out, yield and topology need to be thoroughly understood if reasonable strategies for restructurable logic are to evolve.

TABLE I

1. Routability and wiring strategies for cell interconnection — density and distribution of express and local runs
2. Cell complexity vs. link density tradeoffs
3. Universal cells for customization using local links
4. Electrical characterization of NPL's — area-speed tradeoffs
5. Development of testing procedures based on cell isolation — effects of link and interconnect failures
6. Development of test-contingent, automated routing techniques
7. Design of access decoder for testing and programming links
8. Effects of reducing device dimensions and increasing chip size

Table I, above, shows a summary of eight major areas for substantive research in the area of restructurable logic. At the present time little work has been done in any of these.* However, significant advances have been made in the fabrication technology of MNOS devices, which are now finding their way into a number of consumer markets. It is important to note the strong coupling between the device, circuit and architectural issues involved in the problem areas shown and to recognize that a unified, interdisciplinary approach is required if the promise of restructurable logic is to be realized.

* Since the original submission of this paper, a talk has been given at the International Telemetering Conference in Los Angeles, Cal., on 16 Nov. 1978 entitled, "Wafer integrated semiconductor mass memory," by W.A. Geideman and A.L. Solomon which describes the use of nonvolatile latches to restructure MNOS CCD memory.

References

1. D. Frohman-Bentchkowsky, "The metal-nitride-oxide-silicon (MNOS) transistor characteristics and applications," Proc. IEEE, vol. 58, Aug. 1970.
2. D. Kahng, S.J. Sundburg, D.M. Boulin, and J.R. Ligenza, "Interfacial dopants for dual-dielectric, charge-storage cells," Bell System Tech. J., vol. 53, pp 1723-1739, Nov. 1974.
3. M. Horiuchi and H. Katto, "A low voltage, high-speed alterable n-channel non-volatile memory," IEEE Proc., 1978 Tech. Digest IEDM, pp 336-339.
4. D.L. Greer, "An associative logic matrix," IEEE J. Solid-State Circuits, vol. SC-11, no. 5, Oct. 1976.
5. J.A. Yasaitis, "Ion implantation of neon in silicon for planar amorphous isolation," Electronics Letters 14 (15), 460 (1978).

A SUBNANOSECOND LSI FAMILY FOR MAINFRAME TECHNOLOGY

H. H. Muller, H. Stopper, R. K. Tam

H. H. Muller
BURROUGHS CORPORATION
16701 W. Bernardo Drive
San Diego, California 92127

(714) 487-3000 X4329

ABSTRACT

A subnanosecond LSI family is defined for next generation mainframes. It employs distributed on-chip regulation to reduce system power supply cost, stacked structures for delay-power improvement, on-chip test/diagnostic monitors and signature circuits to improve system maintainability.

SUMMARY

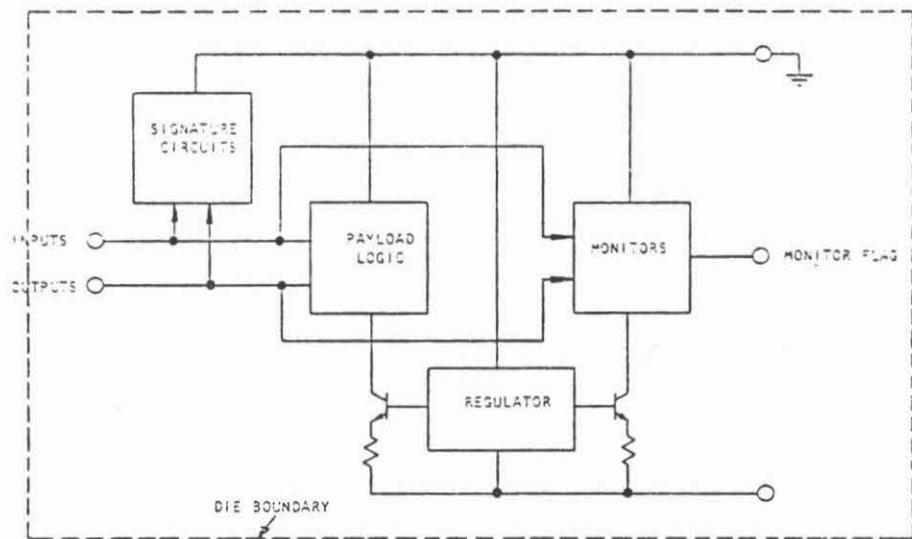
Progression of subnanosecond logic families into large scale integration has opened up new directions and alternatives for high speed mainframe design. The diversity of hardware considerations, however, necessitates an in-depth study to outline optimal configurations for next generation high speed LSI machines. Approaching cost, performance, reliability and maintainability from top down, a 4.65 V+ 18% subnanosecond current switching LSI family has been defined. This LSI circuit family employs an innovative, efficient, low overhead local regulating scheme (Figure 1). It has an efficiency of 97%, occupies an area of less than 200 x 200 μ m for 400mA drive capability. For regulator performance characteristics see Figure 2. Due to the much relaxed power supply tolerance, a 3-phase unregulated power system may be used in conjunction with tolerant distribution, thus resulting in significant cost reduction (Figure 3).

Due considerations have been given to the selection of the supply voltage range so it can accommodate a diversity of circuit types such as RAM and PROM combined with logic elements on one chip. Figure 4 depicts the minimum voltage required for high speed memory and logic. As memory circuits require extra voltage over single level logic gates, an innovative power saver bus (Figure 5) improves the delay-power product of the LSI family. A stacked gate structure fully utilizes the supply voltage for output gates and internal non-series-gated gates.

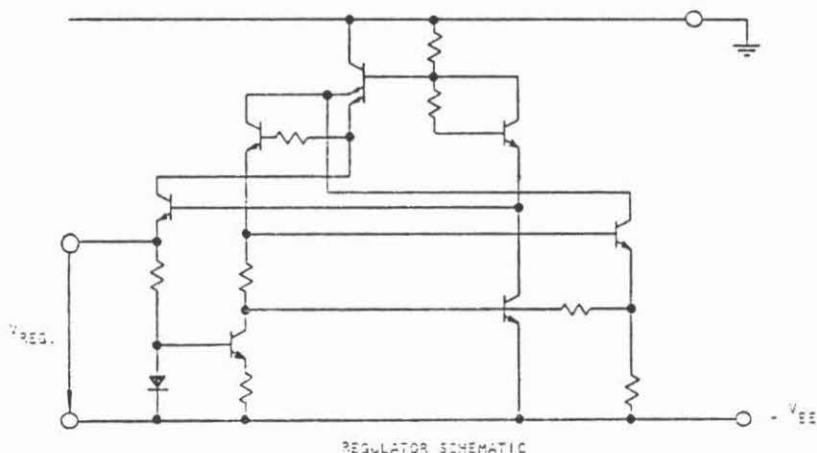
Having defined voltage supply and distribution, and the thermal system conditions, a set of RAM, PROM, payload logic, and monitor circuit cells designed within these boundaries will be described.

All inputs are buffered by emitter followers to reduce AC and DC loading while output structures are designed to serve as source terminations. Discrete active or passive components are neither required nor allowed and the packaging of the LSI system becomes truly homogeneous and reliable. Maintainability is enhanced by implementing on-chip test and diagnostic monitors together with fault isolation facilities. The monitors (Figure 6) assist in factory and field testing and in troubleshooting of the interconnecting signal and power nets with the possibility of an anticipatory maintenance concept. Chip-engraved signatures simplify pseudo random testing of LSI devices even while in-situ. The on-chip signature circuits and their interaction with a hand-held pseudorandom tester will be described.

This LSI philosophy will stimulate advances in mainframe design which demand new concepts in architecture and system partitioning beyond simple cost/performance tradeoffs.



BLOCK DIAGRAM OF LSI DIE CONTENT



REGULATOR SCHEMATIC

FIGURE 1

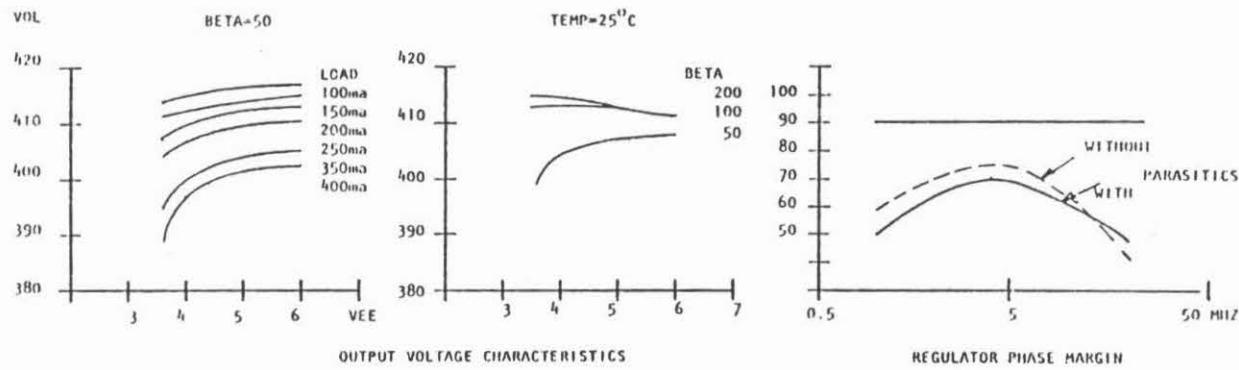


FIGURE 2

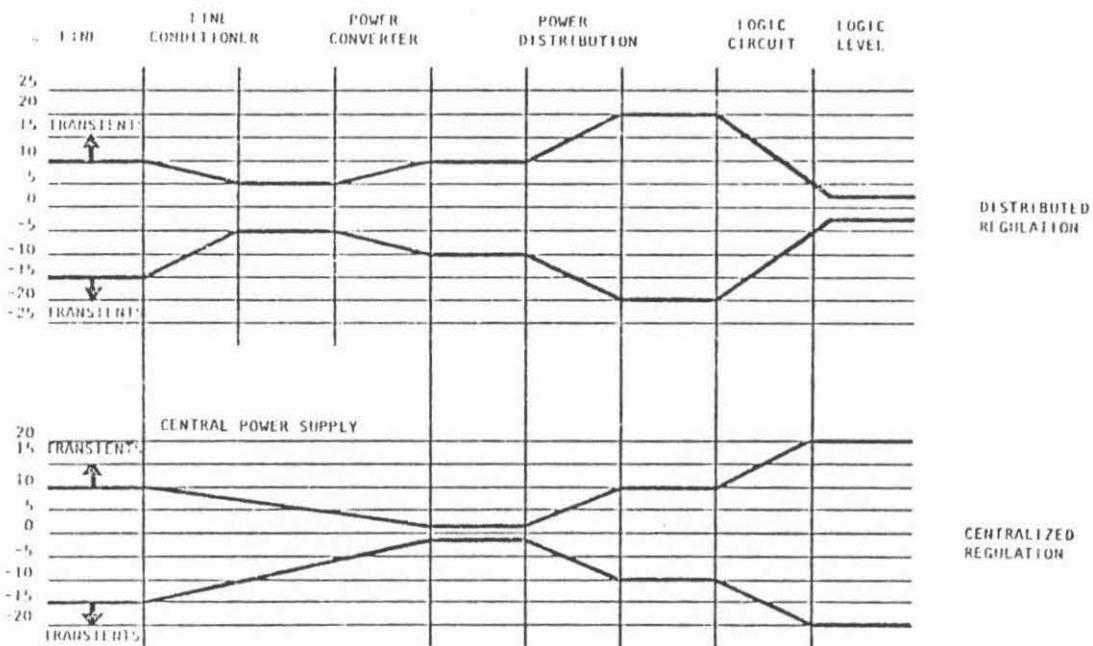


FIGURE 3

CIRCUIT TYPE	MINIMUM VOLTAGE REQUIRED	COMMENT
PROM	3.8	FUSIBLE LINK
LOGIC	3.4 / 3.8	3-4 LEVEL SERIES GATING, E.G., 1/4 AND 1/8 MUX
RAM	3.8	EMITTER COUPLED MEMORY
ROM	1.9/2.8	HIGH/LOW POWER DESIGN

ASSUMPTION:

1.2V VOLTAGE REFERENCE

400MV MAX. FORWARD BIAS ACROSS
BASE COLLECTOR JUNCTION.

FIGURE 4 MINIMUM SUPPLY VOLTAGE FOR CURRENT MODE SWITCHING CIRCUITS

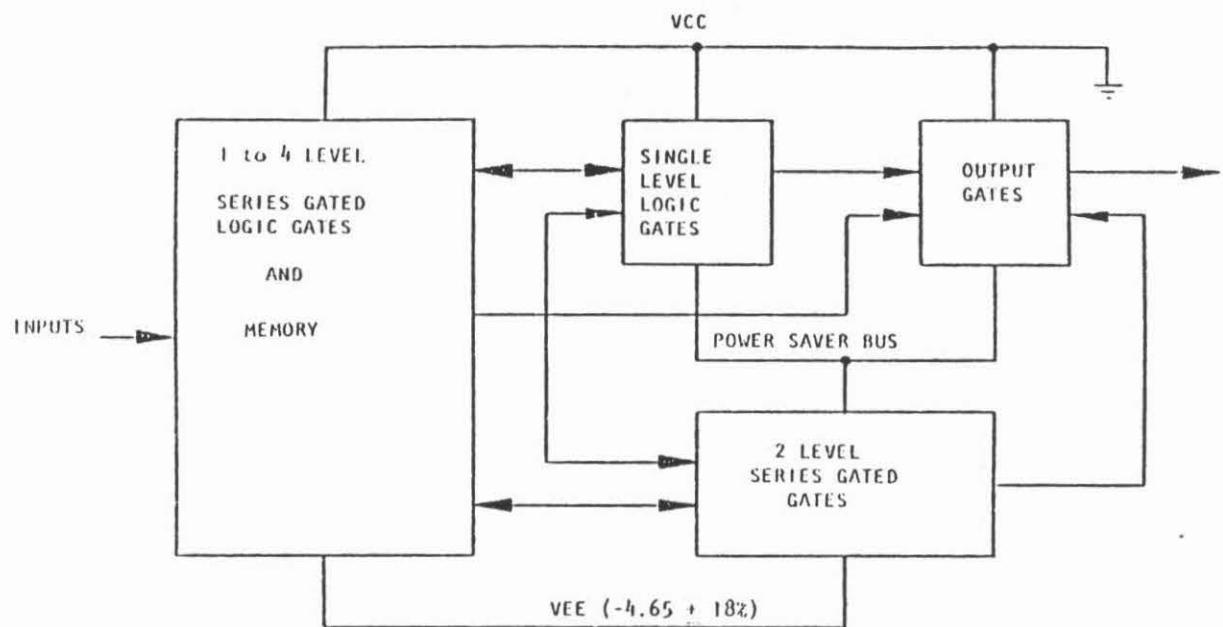


FIGURE 5 SUBNANOSECOND LSI CHIP CONFIGURATION

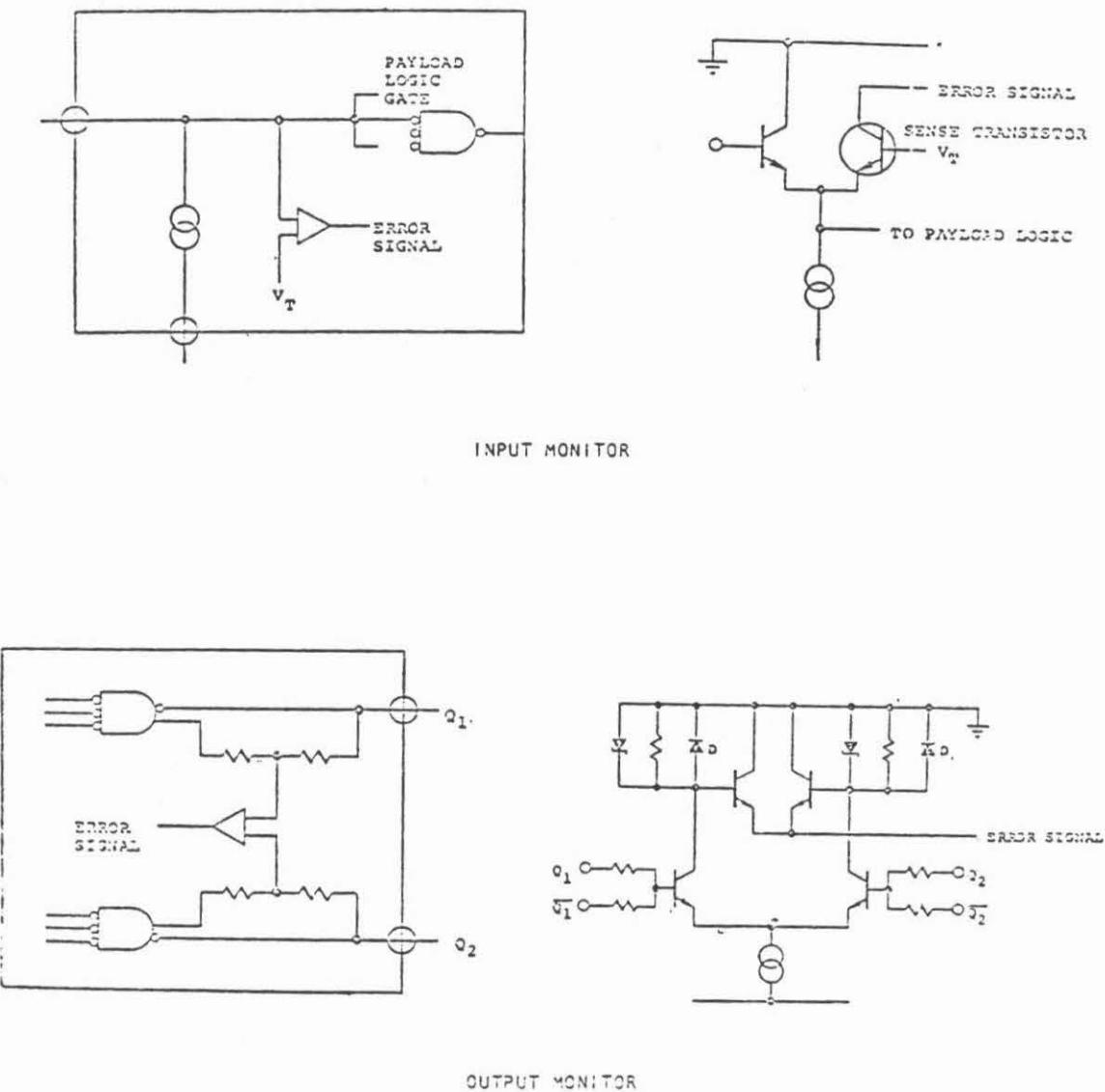


FIGURE 6

A SIMPLE TWO-LAYER ALUMINUM METAL PROCESS FOR VLSI

Robert J. Huber
Electrical Engineering Department
University of Utah
Salt Lake City, Utah 84112

I. Introduction

The use of two levels of metal interconnect lines in an integrated circuit chip layout is a very desirable feature that allows higher density and greater freedom in the placement of the active components. In spite of these benefits it has often been avoided in the design of integrated circuits. For many applications the cost of the extra processing steps is not justified. In the case of MOS technology, long diffusion runs can be successfully used. In silicon gate MOS the polycrystalline silicon itself provides, with some restrictions, a second level of signal interconnect lines. However other technologies, for example I^2L , need a second layer of low-resistance metal interconnect to effectively utilize the chip area. While conceptually simple, two-layer metal processes have proved to be quite difficult to implement. This paper describes a relatively simple two-layer metal process that is well suited to university laboratories and others with limited facilities.

II. Current Practice

The principal problem encountered with two-layer metal processes is due to the surface topography of the insulating layer at the edges of the first layer of metal. The straightforward process for two-layer metal would be as follows. The first layer of metal is deposited and delineated by photoetching in the normal manner. A layer of insulator, most probably silicon dioxide, is then added by chemical vapor

deposition (CVD) on top of first layer metal. Holes are opened through the oxide by photoetching to allow contacts between the two layers. The second layer of metal is then deposited and delineated.

Two problems encountered with this simple procedure make it unworkable. One problem is that the low temperature CVD oxide used with aluminum processes etches very rapidly, making conventional wet etching processes hard to control. The oxide cannot be "densified" at high temperature because of the properties of the silicon-aluminum system. The second and much more severe problem stems from the surface topography of the CVD oxide layer at the edges of the first layer metal. The vapor deposition process, particularly when done at atmospheric pressure, increases the steepness of steps on the surface as shown in Fig. 1. Low angle scanning electron microscope (SEM) examinations of actual structures verify this. Figure 2 is the edge of an aluminum run covered with 5000 Å of CVD silicon dioxide deposited at atmospheric pressure. Addition of phosphorus to the oxide does not help. If anything it makes the edge profile steeper. When second layer metal is deposited on this vertical step, coverage is not good and it often does not survive the etching process as shown in Fig. 3.

Numerous approaches have been taken to solving this problem. Careful control is maintained over the relative thickness of the three layers involved. The CVD oxide and second layer metal will each be about twice as thick as the first layer metal. Such thick layers result in large feature size and create internal stress cracking problems.

A more fundamental approach is taken in those processes that "taper" the edge of the first layer metal. A gentle slope at the edge of first layer metal will be maintained by the CVD oxide. This prevents breaks in the second layer metal. Taper etch processes in general depend on "controlled undercutting" of the photoresist caused by a thin, rapidly-etching layer between it and the metal. One such

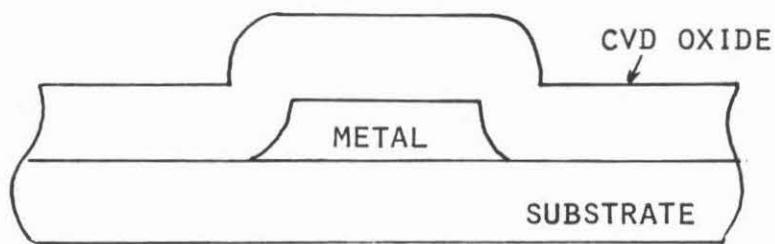


Fig. 1. Cross section of first layer metal covered with CVD oxide.



Fig. 2. SEM photo of edge profile of CVD oxide-covered metal.

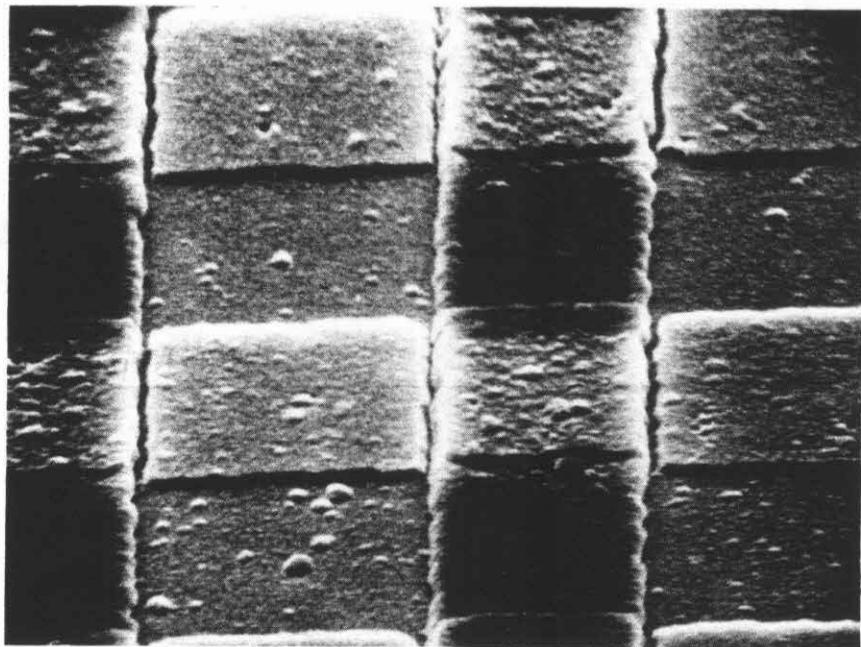


Fig. 3. Metal breaks in second layer metal at edges of first layer metal.

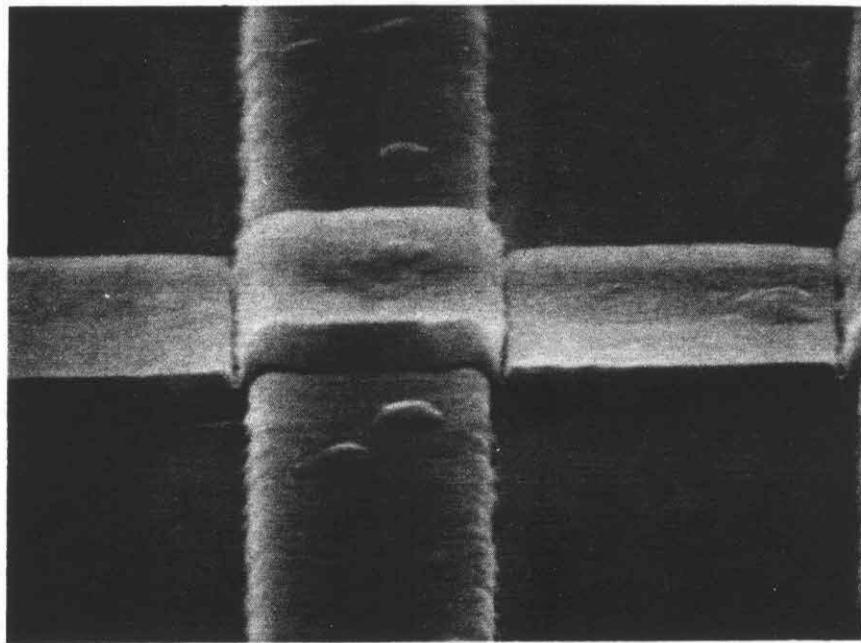


Fig. 4. Inferior metal coverage over sharp edge profile.

process is described by Wilson and Ghate [1].

Other methods improve the metal coverage over the steps by elaborate deposition apparatus. In one commonly used method the metal is sputtered onto high-temperature substrates in a rotating system. Because of the high temperature, the metal atoms have significant mobility once they are on the surface. Obviously this requires expensive equipment.

A direct and successful approach when conditions permit its use is a change in the physical nature of either the interlayer insulator or the process by which it is deposited. If the insulator can be deposited as a liquid, or liquefied after deposition, the surface energy will smooth out the surface. Such an approach is used in the usual silicon gate MOS process. The insulator layer (SiO_2) on top of the patterned polycrystalline silicon is formed by CVD. As deposited it shows the same edge profile problems that are being discussed here. However, it is often deposited in two layers. The top layer contains several percent P_2O_5 , which lowers the softening temperature enough that, following deposition, it can be heated and allowed to flow. The flow eliminates the sharpest surface features and makes possible good metal coverage. Unfortunately this procedure requires temperatures too high to be used with two-layer aluminum.

In the process reported here, the interlayer insulator is an organic polymer which is deposited as a liquid using a conventional photoresist spinner. It cures to a polyimide which can withstand the temperatures [2] encountered in the wafer processing that follows second layer metal and in the die-attach, wire-bond, and package-sealing operations.

III. Polyimide Film as Interlevel Insulator

Several researchers have reported on the use of polyimide in planar structures. Among them are Sato, et al. [3] who describe a

multilayer metal interconnection technology in which holes for the interlayer electrical contacts are opened by a uniform removal of the polyimide down to "bumps" in the first layer metal. More recently Yen [4] described a low-cost polyimide interlayer insulation process. A polyimide passivation reliability study was reported by Gregoritsch at the 1976 Reliability Physics Meeting [5], and it has been reported that IBM Corporation is using a polyimide interlevel insulation in a new 65 K RAM chip [6].

IV. Polyimide Photoresist Two-Layer Metal Process

This process uses a photoresist* [7] as an interlayer insulator which contains sensitizers and precursors to polyimide. Following exposure and development, proper curing steps convert it to a polyimide which is thermally stable above 400°C. Because the interlayer insulator is deposited as a liquid, it gives a surface profile that is smoother than exists under it. As it is also a photoresist, no additional layers must be used to photoetch via holes. The process as developed in these experiments follows:

1. Finish wafers through first layer aluminum metal using the standard process. The metal must be given an alloy at as high a temperature as any of the subsequent processing.
2. Deposit by CVD methods about 1000 Å of SiO₂. Use of this oxide was found to give better adherence of the polyimide than did the aluminum alone.
3. Treat the surface with a coupling agent to promote adherence of the polyimide-based photoresist. We used hexamethyldisilizane (HMDS) diluted in Freon TF (1-1-2 trichlorotrifluoroethane) in the ratio of two parts HMDS to one part Freon TF, applied to the wafer on a conventional photoresist spinner at 2500 rpm.

* These experiments used PR-514, a product of GAF Corporation. Since this work was done, that organization sold its photoresist business. It is not known at this time if this particular photoresist will continue to be available.

4. Immediately coat the wafer with the photoresist on the same spinner at 2000 rpm. This speed gives a layer approximately 1.4 μm thick.
5. Bake in nitrogen one hour at 80°C.
6. Align mask and expose. The resist is not very sensitive and about one minute UV exposure is required. Resist is positive working.
7. Develop 20 seconds in the developer supplied by the manufacturer. Rinse in deionized water and air dry for 15 minutes.
8. Reexpose the remaining photoresist without a mask to decompose any remaining sensitizer. Exposure should be at least twice the preceding exposure.
9. Bake the wafer in a series of steps beginning at 140°C for 15 minutes, and increasing the temperature about in 50°C increments, finishing at 440°C in N_2 for 5 minutes. The lower temperature bakes were done on a hot plate while the 440°C temperature bake was in a standard diffusion furnace.
10. Etch the 1000 Å of CVD SiO_2 , which is now exposed under the open via holes, in standard buffered HF-based etch. Do not overetch.
11. Deposit second layer metal and finish the wafers normally.

After the polyimide has been exposed to 440°C it is not appreciably attacked by the usual photoresist strippers so it needs no protection during the photosteps which delineate second layer metal.

V. Results

This procedure was evaluated using a test pattern containing many metal crossovers and on an integrated circuit built with two-layer metal.

A comparison of the quality of metal crossovers obtained with polyimide and with CVD SiO_2 is shown in Figs. 4 and 5. Figure 4 shows a crossover obtained with the low temperature CVD SiO_2 and electron beam evaporated aluminum. Figure 5 shows a crossover fabricated from

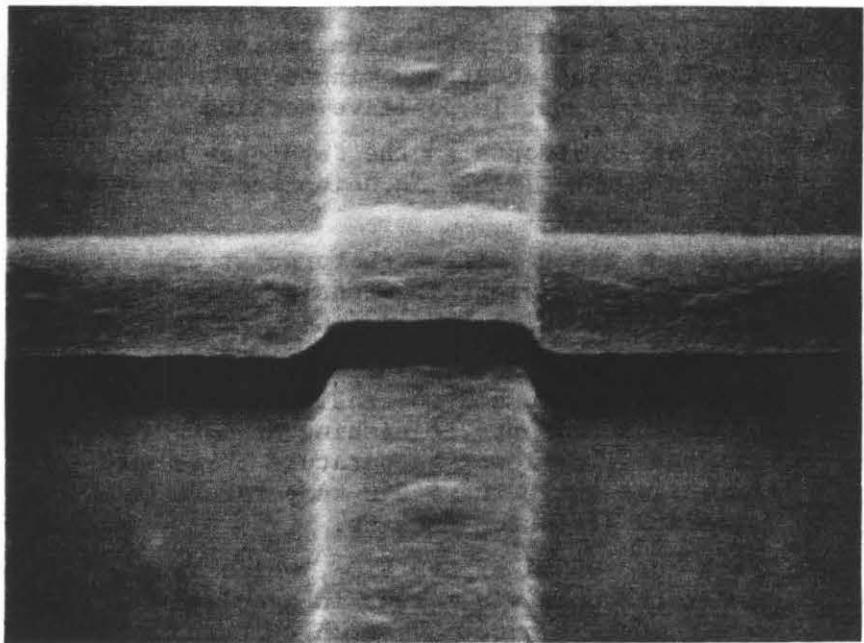


Fig. 5. Improved metal coverage over polyimide-covered first layer metal.

the same set of masks using the photoresist procedure described here. Note the much improved coverage where second layer metal crosses first layer metal. The advantage of the polyimide-covered surface is the more gentle slope at the edges of first layer metal.

A two-layer metal pattern containing 600 crossovers was built to check the incidence of metal breaks over the steps, the inter-level resistance and the ability of the polyimide to withstand thermal stress. The total area in which the two layers of aluminum are separated by polyimide was $6 \times 10^{-4} \text{ cm}^2$. The incidence of metal breaks was extremely low. Too few breaks were found in these experiments to allow statistically meaningful yield predictions.

The structure was subjected to temperature cycling between $+150^\circ\text{C}$ and -40°C . First the pattern was heated to 150°C in air for one week with a constant 10 volts applied between the layers. Resistance between the layers was steady at $2 \times 10^{11} \Omega$ at 150°C . Then the structure was cycled between -40°C and $+150^\circ\text{C}$ six times with the interlayer resistance measured at each temperature extreme with 10 volts applied. At -40°C interlevel resistance was $2.5 \times 10^{13} \Omega$ and at $+150^\circ\text{C}$ interlevel resistance was $2.3 \times 10^{11} \Omega$. Heating and cooling rates were about 40 degrees/minute. At the end of the tests, all metal runs were still continuous and no interlevel shorts observed.

A simple 1024-bit I^2L read-only-memory, organized as a 16×64 array, was built using this two-layer metal process to verify the ability of this process to actually produce an LSI circuit. A functionally good chip was placed on high temperature life test at 100°C and normal operating voltages and signals applied. Operation continued uninterrupted for over 6000 hours with no apparent degradation.

VI. Discussion and Conclusions

A two-layer metal process suitable for experimental two-layer aluminum metal has been described. It requires no additional equipment

over that required for the simplest single-layer aluminum and CVD SiO_2 processes. The interlevel insulator is a dual layer consisting of 1000 Å of CVD SiO_2 and about 1 μm of polyimide. The polyimide is obtained from a positive working polyimide-based photoresist. Interlayer contact holes are formed by exposure and development of the layer when still in the photoresist form, thereby eliminating many process steps used for polyimide layers that are deposited in the pure form. The polyimide shows no degradation when exposed to temperatures of 440°C in nitrogen and therefore circuits incorporating this layer will withstand normal packaging operations.

Acknowledgment

This work was carried out at the University of Utah Research Institute Microcircuits Laboratory with support from General Instrument Corporation.

References

1. A. M. Wilson and P. B. Ghate, "A Two-Layer Metal System Designed for Large-Scale Integrated Circuits", in *Semiconductor Silicon 1977*, edited by H. R. Huff and E. Sirth, Electrochemical Society, Princeton, New Jersey, 1977.
2. S. D. Bruck, "Thermally Stable Polymeric Materials", *Journal of Chemical Education*, Vol. 42, No. 1, January 1965, p. 19.
3. K. Sato, et al., "A Novel Planar Multilevel Interconnection Technology Utilizing Polyimide", *IEEE Transactions on Parts, Hybrids, and Packaging*, Vol. PHP-9, September 1973, pp. 176-180.
4. J. C. Yen, "A High Yield and Low Cost Process for Building Multi-layer Metal Structures by Using PYRE-NIL", *Electrochemical Society Fall Meeting, Extended Abstracts*, October 5-10, 1975, pp. 444-445.
5. A. J. Gregoritsch, "Polyimide Passivation Reliability Study", *Proceedings of the 14th Annual International Reliability Physics Symposium*, Las Vegas, Nevada, April 20-22, 1976, Electron Devices Society and IEEE Reliability Group.

6. "IBM Shows Off an Unusual New Family of Dynamic RAMs", *Electronics*, November 9, 1978, p. 39.
7. A. J. Battisti and F. J. Loprest, "A New High Temperature Positive Working Photoresist", presented at SEMICON/East, Uniondale, New York, September 17, 1975.

A COMPUTER STUDY OF ELECTRON-ELECTRON INTERACTION
IN HIGH DENSITY ELECTRON BEAMS

Tateaki SASAKI

The Institute of Physical and Chemical Research
Wako-Shi, Saitama 351, Japan

and

Department of Computer Science
The University of Utah
Salt Lake City, Utah 84112, USA

ABSTRACT

High density electron beams are simulated by a computer, and the trajectory displacement and energy broadening caused by electron-electron interaction are investigated computationally. The results are summarized into two empirical formulas which represent dependences of the average trajectory displacement and the average energy broadening on the beam parameters. The results show that the trajectory displacement caused by electron-electron interaction imposes a severe problem on system designers using high density beams, and that energy broadening on the order of 1eV may well be attributed to electron-electron interaction. The method of simulation is also described.

1. Introduction

Use of high density electron beams seems to be very promising for fast VLSI pattern exposure, and several groups have already proposed VLSI-oriented pattern exposure schemes using high density beams [1,2,3,4].

Using high density beams, however, we can not ignore the effect of the Coulomb interaction between beam electrons (so-called electron-electron interaction). In fact, many experiments have revealed that this effect becomes remarkable when the beam current reaches $1\mu A$ [5,6,7]. Since this effect weakens beam focusing and reduces the resolution of the beam, system designers should be aware of such properties if they want to use high density beams.

Because of its importance, many authors investigated the effect of electron-electron interaction theoretically [8,9]. However, it seems, at least for the present author, that the effect is not well clarified yet. One reason is that the phenomenon is a many-body effect and not easy to investigate theoretically. Another reason is that actual beam geometries are complicated, making the analytical calculations difficult. These reasons stimulate us to simulate the phenomenon by computers and study the effect computationally.

This paper describes one such study performed recently. In this work, electron beams very close to actual beams were simulated by a computer, except that the numbers of electrons treated were much smaller than those in actual cases. Since the Coulomb force is long-range, we must take forces between many pairs of electrons into consideration in the simulation. This prevents us from treating more than several thousand electrons at one time by even a fast computer. However, error analysis shows that the accuracy of our results is about 5%.

Electron-electron interaction was investigated computationally by Loeffler and Hudgin [10] about ten years ago. However, their method is considerably different from ours. They first generated unperturbed electron trajectories randomly in which electron-electron interaction was neglected. Then, they estimated the effect of electron-electron interaction by assuming electrons to move along their unperturbed trajectories. In our simulation, motions of electrons are traced by a small time step by taking electron-electron interaction into the equation of motion at each time step. Furthermore, we treat beams having many crossovers, while Loeffler and Hudgin treated beams having only one crossover.

2. Assumptions in simulation

The equation of motion of the i -th electron is

$$\ddot{m}_e \vec{r}_i = \vec{F}_{ext} + \frac{e^2}{4\pi\epsilon_0} \sum_{j \neq i} \frac{\vec{r}_i - \vec{r}_j}{|\vec{r}_i - \vec{r}_j|^3}, \quad i=1,2,3,\dots,$$

where m_e is the electron mass, \vec{r}_j is the coordinate vector of the j -th electron, and \vec{F}_{ext} is an external force. The second term in the right side represents electron-electron interaction. We discard the force due to a magnetic field caused by the beam current because we treat only non-relativistic beams. The problem is to solve the above coupled non-linear equations with some initial conditions.

The first assumption we make in our simulation is that electrons are ideally accelerated in such a way that they have a Maxwellian velocity distribution corresponding to the cathode temperature when they are observed in a moving system in which the average electron velocity is zero. This means that we neglect the effect of electron-electron interaction during the acceleration. Although this assumption was adopted in many theoretical calculations, its validity is an open problem. We also assume that the electrons are accelerated until they reach the first crossover and no acceleration field acts on them behind the first crossover.

The second assumption is that the momentum of an electron after the acceleration is much greater than the momentum which the electron receives during its path from the gun to the screen by electron-electron interaction. Since the kinetic energy of an electron after the acceleration is very large in actual beams, this assumption is quite valid. With this assumption, we can treat the electron-electron interaction term as a perturbation. Unperturbed trajectories are those which are determined only by \vec{F}_{ext} .

The third assumption is that we can discard interaction between electrons which are very distant from each other. This assumption is necessary because of limited computational power, and its validity is checked computationally.

We simulate beams of circular cross sections which are focused by a uniform coaxial magnetic field with a focal length of 5cm. A uniform focusing magnetic field is very convenient for simulation because we can calculate unperturbed motions of electrons exactly. Furthermore, uniformity of the external force makes the programming quite simple. Note that the motions of electrons near crossovers, where electron-electron interaction is the most effective, are not so dependent on whether the focusing is made by a uniform magnetic field or a combination of electric and magnetic fields. Note further that, in our

scheme, we can simulate nearly divergent beams by only weakening the focusing magnetic field.

Following Loeffler and Hudgin [10], the effect of electron-electron interaction may be divided into three parts, the trajectory displacement $\Delta\vec{r}$, the energy change ΔE , and the angular change $\Delta\alpha$. However, we consider only the trajectory displacement and the energy change because the angular change is not so important in actual system designs.

3. Method of simulation

We characterize an electron beam by the beam current I , the electron acceleration voltage V , the beam radius r_C at the first crossover, the beam semi-angle α at the first crossover, the beam length L , and the cathode temperature T .

Following the assumptions given in a previous section, we emit N electrons randomly at the gun (precisely speaking, at the first crossover) in such a way that they have the beam parameters I , V , r_C , α , and a Maxwellian energy distribution with the temperature T . Therefore, we are not treating beams ranging from the gun to the screen but a group of electrons. The size of the group is only about several millimeters.

These electrons are moved toward the screen discretely by the time step $(L/\bar{v})/n$, where \bar{v} is the average axial velocity of the electrons and n is a large integer. At each time step, the motion of each electron, except for m electrons at each end of the group, is first determined by the focusing magnetic field and then corrected by calculating the forces from the nearest $2m$ electrons. The correction is performed only for those electrons which have not yet passed through the screen. Therefore, the number of Coulomb force evaluations in one simulation is about $2m(N-2m)n$.

When all electrons reach the screen, we calculate the trajectory displacement $|\Delta\vec{r}|$ at the place of screen and the energy change $|\Delta E|$ of each of the central $N-4m$ electrons in the group by comparing its motion with its unperturbed motion. The $2m$ electrons at each end of the group are discarded so as to reduce the "end effect." In this way, we get distributions of the trajectory displacements and the energy changes of $N-4m$ electrons.

Let us call the values of the beam parameters "standard" when they are $I=2\mu A$, $V=20kV$, $r_C=10\mu m$, $\alpha=2mrad.$, $L=30cm$, and $T=2500^{\circ}K$. By changing each beam parameter with others fixed to the standard values, we simulated many beams under different conditions. In this way, results were obtained for the dependences of the average trajectory displacement $\langle|\Delta\vec{r}|\rangle$ and the average energy change $\langle|\Delta E|\rangle$ on the beam parameters. The range of beam parameters investigated are $0.5\mu A \leq I \leq 10\mu A$, $5kV \leq V \leq 100kV$, $5\mu m \leq r_C \leq 20\mu m$, $5cm \leq L \leq 50cm$, $1mrad. \leq \alpha \leq 10mrad.$, and $500^{\circ}K \leq T \leq 2500^{\circ}K$.

The simulation parameters, i.e., N, n, and m, were set in most cases as N=1200, n=1000, and m=50. We checked appropriateness of the values of m and n in beams of length 5cm with other beam parameters fixed to the standard values. We found that, if we changed the value of m from 50 to 100, the value of $\langle |\Delta f| \rangle$ increased by about 2.2% and the value of $\langle |\Delta E| \rangle$ decreased by about 3.7%. If we changed the value of n from 100 to 200, the values of $\langle |\Delta f| \rangle$ and $\langle |\Delta E| \rangle$ increased by about 0.4% and 0.6%, respectively. We estimated the amounts of accumulated rounding errors by comparing unperturbed trajectories calculated analytically with those calculated by our simulation program with the time step n=1000. We found that the errors caused by rounding are less than 0.05%. We may conclude from these results that the errors come mainly from smallness of m (2.5~3.5%) and N ($1/\sqrt{1000} \sim 3.2\%$), hence the accuracy of our results is about 5%.

4. Results of simulation

As we have mentioned above, two distributions are obtained in each simulation: One is of trajectory displacements and the other is of energy changes. These distributions are similar to a Gaussian and an exponential distributions with broadened tails, respectively. For each distribution, we calculate the average and the variance.

Figures 1 to 6 show our results of simulations. In these figures, we used the term "energy broadening" instead of the term "energy change." Throughout these figures, a black dot and a white circle denote average values of the trajectory displacement and the energy change, respectively. Let the average and the variance of a distribution be f and v , respectively. In each figure, a vertical line (solid or dashed) represents the range of a distribution: The top and the bottom of such a line represent $f+v$ and $f-v$, respectively. For ease of readability, a smooth solid curve is fitted to black dots and a smooth dashed curve is fitted to white circles.

We fitted a simple fractional power curve to each parameter dependence. By neglecting the constant terms, the result may be expressed as follows:

$$\langle |\Delta f| \rangle \propto L I V^{-4/3} r_c^{-1/5} \alpha^{-3/4} T^{-1/10}, \quad \text{for } \alpha < 5 \text{ mrad.},$$

$$\langle |\Delta E| \rangle \propto L^{1/2} I^{1/2} V^0 r_c^{-1/3} \alpha^{-2/5} T^0, \quad \text{for } \alpha < 5 \text{ mrad.}$$

The α dependence tends to be very small for $\alpha > 5 \text{ mrad.}$

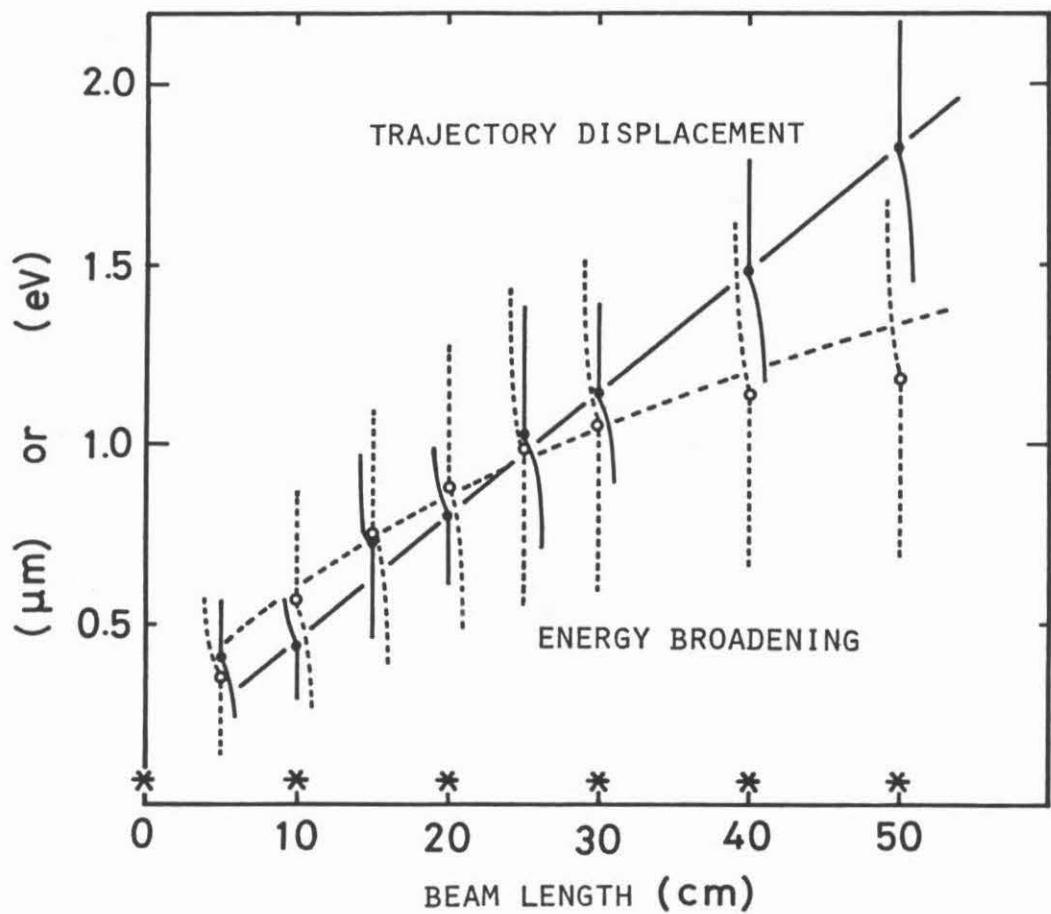


Fig.1 Dependence on the beam length L .
 Other beam parameters are $I=2\mu\text{A}$, $V=20\text{kV}$,
 $r_c=10\mu\text{m}$, $\alpha=2\text{mrad.}$, and $T=2500^\circ\text{K}$.
 The asterisks denote crossover points.

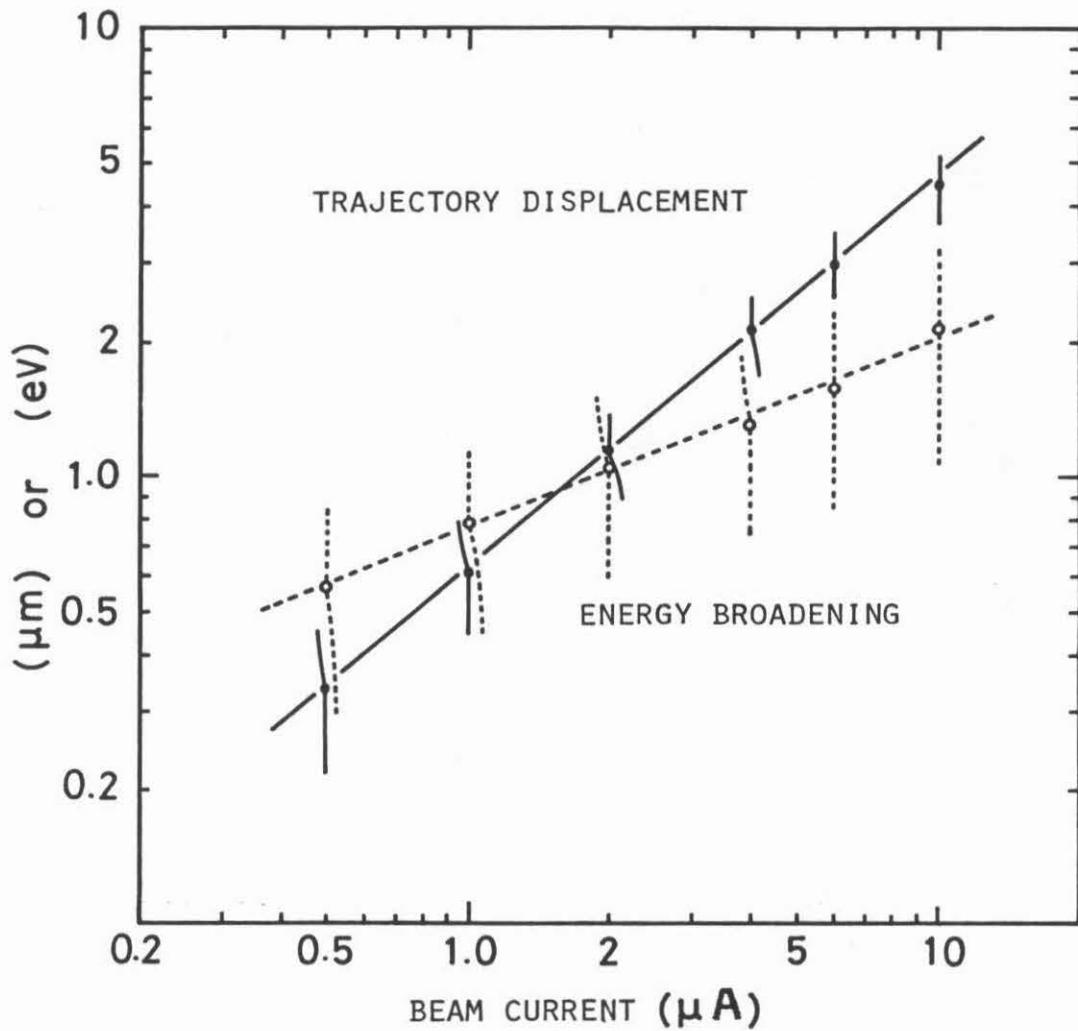


Fig.2 Dependence on the beam current I .
Other beam parameters are $V=20\text{kV}$, $r_c=10\mu\text{m}$,
 $\alpha=2\text{mrad.}$, $L=30\text{cm}$, and $T=2500^\circ\text{K}$.

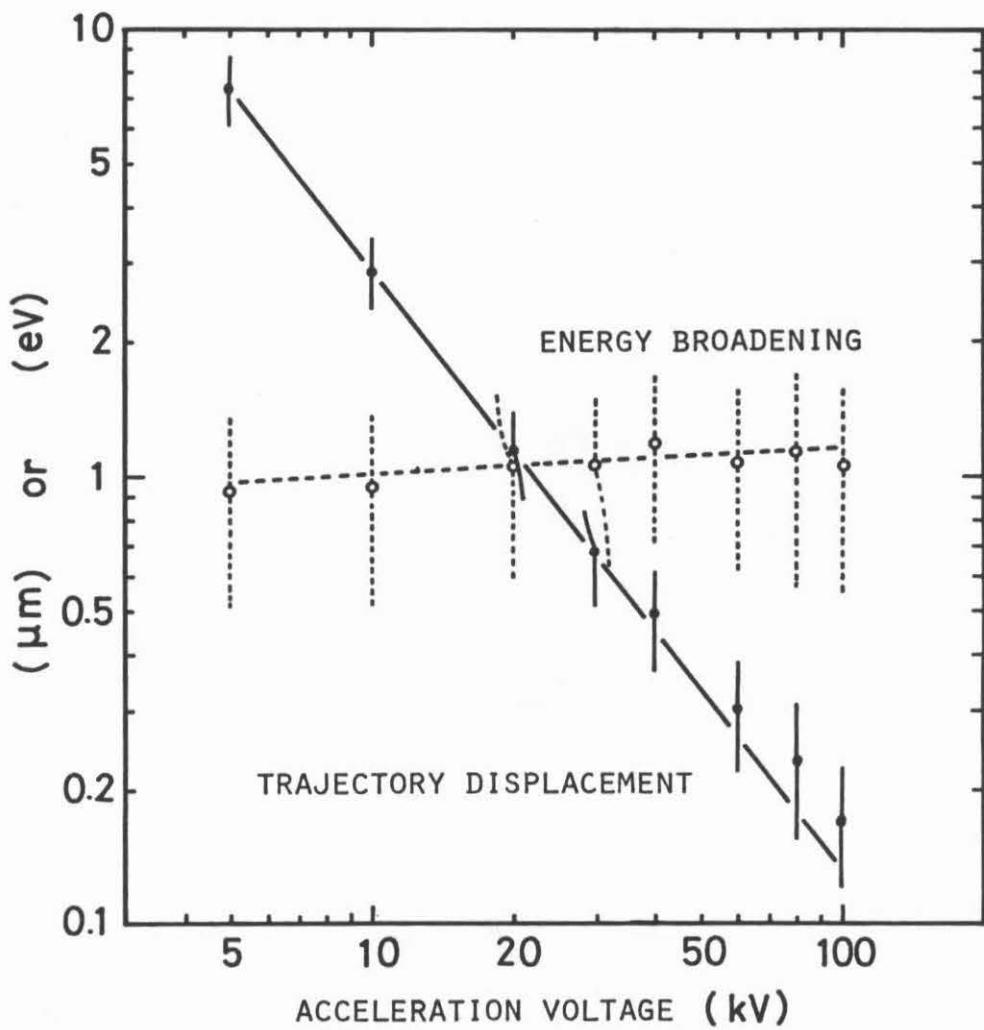


Fig.3 Dependence on the acceleration voltage V .
Other beam parameters are $I=2\mu\text{A}$, $r_c=10\mu\text{m}$,
 $\alpha=2\text{mrad.}$, $L=30\text{cm}$, and $T=2500^\circ\text{K}$.

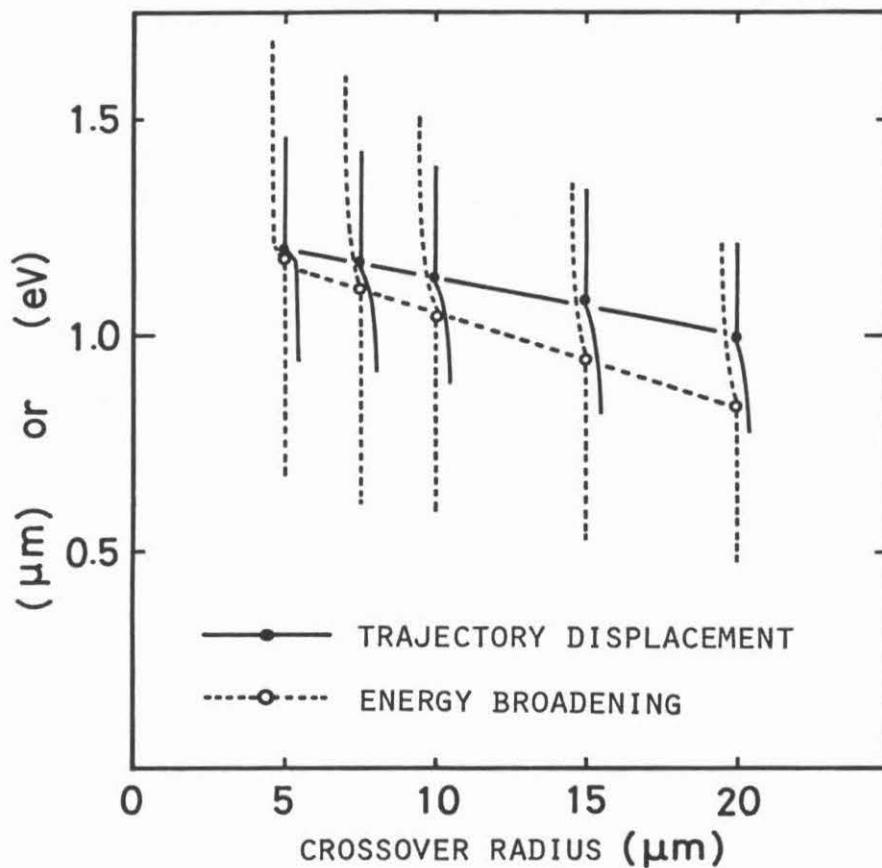


Fig.4 Dependence on the crossover radius r_c .
Other beam parameters are $I=2\mu\text{A}$, $V=20\text{kV}$,
 $\alpha=2\text{mrad.}$, $L=30\text{cm}$, and $T=2500^\circ\text{K}$.

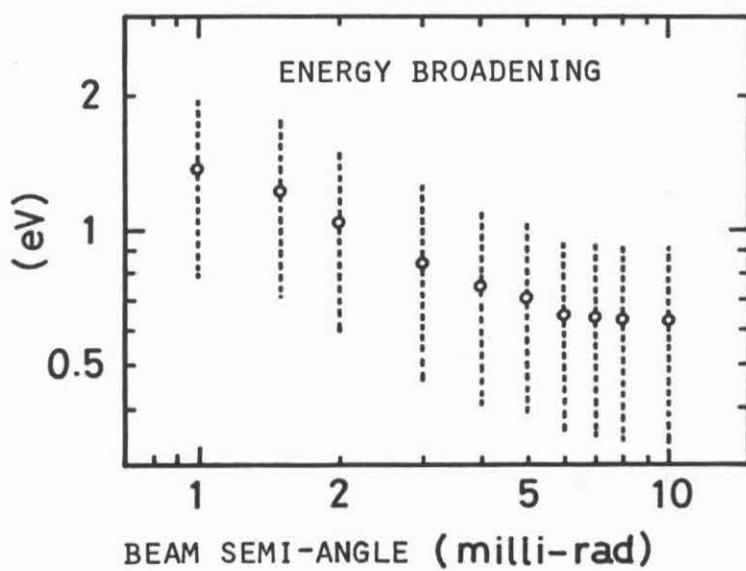
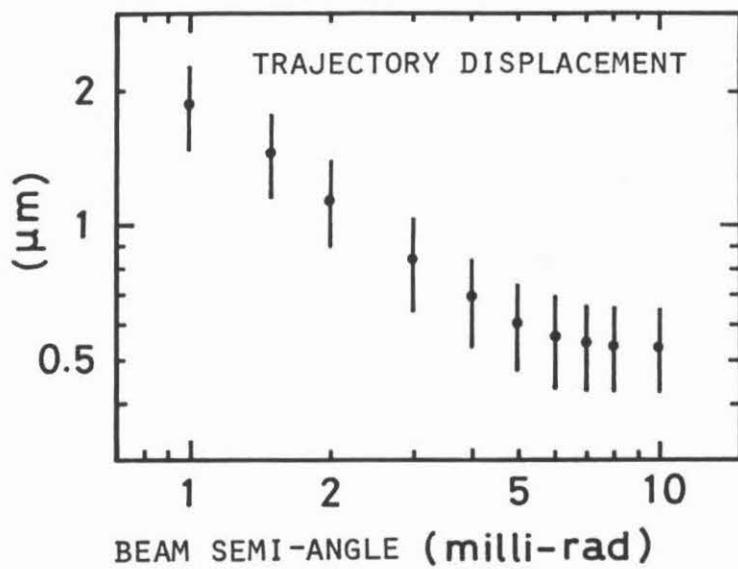


Fig.5 Dependence on the beam semi-angle α .
 Other beam parameters are $I=2\mu\text{A}$, $V=20\text{kV}$,
 $r_c=10\mu\text{m}$, $L=30\text{cm}$, and $T=2500^\circ\text{K}$.

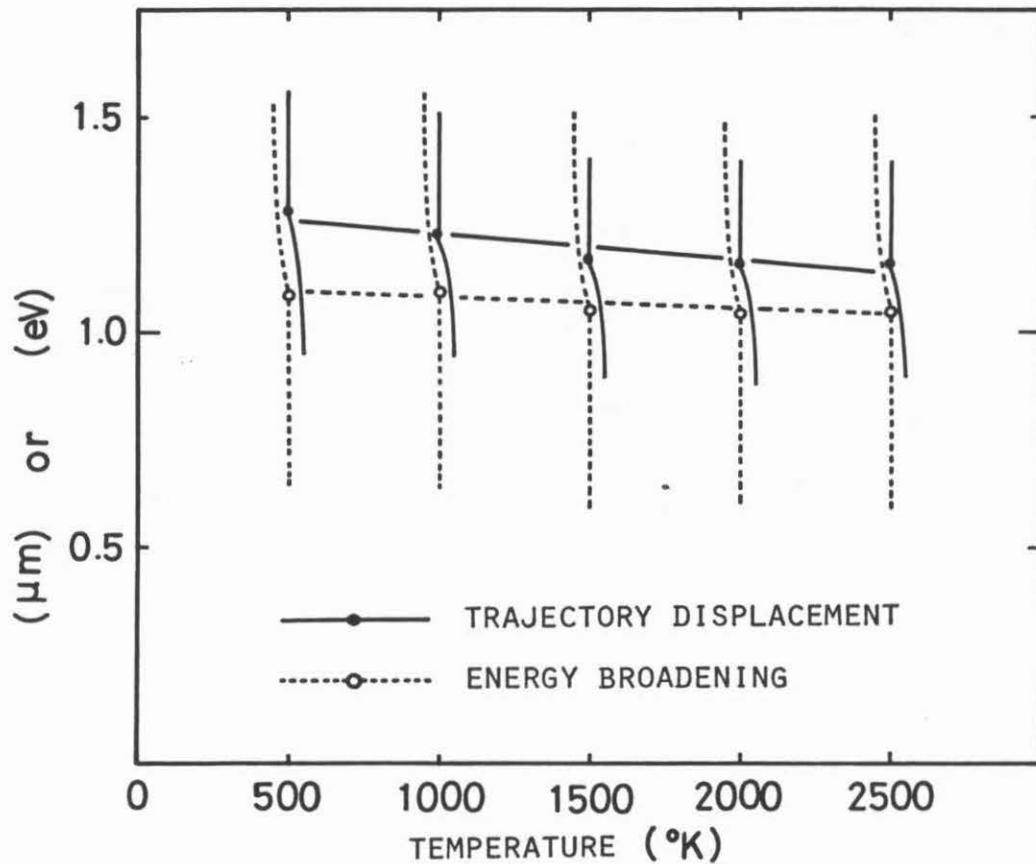


Fig.6. Dependence on the cathode temperature T .
Other beam parameters are $I=2\mu\text{A}$, $V=20\text{kV}$,
 $r_c=10\mu\text{m}$, $\alpha=2\text{mrad.}$, and $L=30\text{cm}$.

5. Discussions and conclusions

Our results show that, when the beam parameters are "standard," the average trajectory displacement and the average energy broadening are about $1\mu\text{m}$ and 1eV , respectively. This result is consistent with an experiment [11].

Since electron-electron interaction in electron beams is stochastic, it is very difficult to eliminate the aberration caused by it. However, we can eliminate some parts of trajectory displacements dynamically. The electric field caused by beam electrons has a smoothed part which is produced by an equivalent smeared-out electron distribution. This field causes systematic trajectory displacements, and most parts of which can be eliminated by refocusing. We found that, when the beam parameters were standard, about half of $\langle|\Delta\vec{r}|\rangle$ was eliminated by refocusing.

We compared our calculations with those of Loeffler and Hudgin by simulating nearly divergent beams of length $L \leq 10\text{cm}$ with other beam parameters fixed to the standard values. We found that our value of $\langle|\Delta\vec{r}|\rangle$ was about two times greater than theirs and our value of $\langle|\Delta E|\rangle$ was about four-fifths of theirs. Considerable parts of these differences are explainable by the difference in beam geometries: Our beams start at the first crossover point while their beam geometries are symmetric with respect to the crossover point. However, several remarkable differences are observable from the dependences on beam parameters. A complete discussion on these differences as well as comparisons with theories will be presented elsewhere.

After Boersch [5] found that the energy distribution of electrons at the screen was broadened, there were many discussions on energy broadening, including critical experiments [12]. Our results, however, show clearly that energy broadening on the order of 1eV may well be attributed to electron-electron interaction.

From the viewpoint of system designers using high density electron beams, trajectory displacement is much more important than energy broadening, as was pointed out by Pfeiffer [13]. In fact, the trajectory displacement on the order of $1\mu\text{m}$, which according to our results is common in high density beams such as proposed in [1] or [2], is never allowable in VLSI fabrication.

Studying phenomena by computers is very popular and playing an important role in areas of plasma physics etc. Compared with most simulations in such areas, simulations of electron beams are quite easy because the effect of nonlinear terms in the equation of motion is quite small. In this sense, the author would like to emphasize the usefulness of computer study of electron beams.

ACKNOWLEDGEMENTS

The author thanks Prof. E. Goto of the Institute of Physical and Chemical Research (IPCR), where most of this work was performed, for initially suggesting this work, and for valuable discussions. He also thanks Dr. T. Soma of IPCR for useful discussions and several references, and Messrs. S. Miyauchi, K. Tanaka and T. Someya of Japan Electron Optics Laboratory Co. for experimental results. Thanks are also due to the members of Department of Computer Science, the University of Utah, where this paper was written, for their hospitality, in particular to Prof. A.L. Davis for correcting English in the manuscript.

REFERENCES

- [1] E. Goto, T. Soma, and M. Idesawa, "A design of a variable aperture projection and scanning system for electron beam," 14th Symp. on Electron, Ion, and Photon Beam Technology, May 1977.
- [2] H.C. Pfeiffer, "Variable spot shaping for electron beam lithography," *ibid.*, 1977.
- [3] M.G.R. Thomson, R.J. Collier, and D.R. Herriott, "A double aperture method of producing variably shaped writing spots for electron lithography," *ibid.*, 1977.
- [4] J. Trotel, "E-beam dynamic shaping," *ibid.*, 1977.
- [5] H. Boersch, "Experimentelle Bestimmung der Energieverteilung in thermisch ausgelosten Elektronenstrahlen," *Z. Phys.*, 139, p.115, 1954.
- [6] H.C. Pfeiffer, "Experimental investigation of energy broadening in electron optical instruments," 11th Symp. on Electron, Ion, and Laser Beam Technology, 1971.
- [7] R.W. Ditchfield and M.J. Whelan, "Energy broadening of the electron beam in the electron microscope," *Optik* 48, p.163, 1977.
- [8] K.H. Loeffler, "Energy-spread generation in electron-optical instruments," *Z. Angew. Phys.* 27, p.145, 1969.
- [9] B. Zimmermann, "Broadened energy distribution in electron beams," *Adv. Electronics and Electron Physics* 29, p.251, Academic Press, 1970.
- [10] K.H. Loeffler and R.H. Hudgin, "Energy-spread generation and image deterioration by the stochastic interactions between beam electrons," *Septième Congrès Int. Microscopie Électronique*, 1970.
- [11] G. Cogswell, S. Miyauchi, K. Tanaka, and N. Goto, "Variable shaped electron beam lithography," 1978 Meeting of Electrochemical Society, May 1978.
- [12] T. Ichinokawa, "Effect of electron source to energy resolution in electron velocity analysis - Interpretation of Boersch effect," *Japanese J. Appl. Phys.* 8, p.137, 1969.
- [13] H.C. Pfeiffer, "Basic limitations of probe forming systems due to electron-electron interaction," 5th Symp. on Scanning Electron Microscope, 1972.

IMPLEMENTING VLSI SYSTEMS IN A RESEARCH ENVIRONMENT

ROBERT W. HON

DEPARTMENT OF COMPUTER SCIENCE
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PA. 15213

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Traditionally, integrated circuits have been designed and implemented only for applications likely to require large numbers of chips. The IC's were realized by closely cooperating experts, each well-versed in some aspect of the art of integrated circuit design and manufacturing. Economics and the detailed knowledge needed prohibited any group unwilling or unable to make a considerable resource commitment from producing integrated circuits. Prototypes and one- or few-of-a-kind systems were built from off-the-shelf components (or not at all). Recently, advances in IC technology and the emergence of new design methodologies have made it possible for people lacking IC manufacturing expertise to design VLSI circuits. While the population able to cast their ideas in silicon is increasing, a previously unseen set of problems has emerged. The remainder of this paper will examine the background of these people and some of the problems that are now critical obstructions in the path to implementing VLSI circuit designs.

The new group of integrated circuit designers comes primarily from research organizations at universities or industrial firms. It is important to stress that they are not in the business of producing IC's for sale, rather, they are building experimental or prototype systems. Members of the group may have physics or engineering or computer science as a main field of research. Their reasons for designing a VLSI system range from the need to implement a particular function for a real-time application to testing the feasibility of a new machine architecture. Designing at the level of individual devices allows these researchers to adjust circuit complexity / execution speed tradeoffs to suit their needs. They also gain the freedom to employ novel or clever structures which might be particularly well-suited to solving the problem at hand. For these designers VLSI is a particularly effective means to an end. At the same time, designs which may conceivably be built in a commercially available

technology lend credibility to such prototyping efforts.

In the semiconductor industry, an integrated circuit ultimately destined for mass manufacturing and sale is created in a multi-step process which involves a number of people with diverse backgrounds. Engineers or computer scientists produce a block diagram of a new function (or processor, or whatever) that they want. After a logic diagram is made, other engineers do the detailed electrical circuit design, keeping in mind the devices available to them in the particular process (e.g. CMOS). Layout specialists then produce specifications for a set of masks to be used in the selective patterning of the layers on the silicon wafer. Once the mask specifications are completed the actual masks are generated in a process which is essentially photographic. From there, the masks are used by the fabrication line to pattern the surface of silicon wafers each containing several hundred copies of the IC. The wafers are separated into individual chips (dicing), mounted in IC packages, and wires are connected from the pads on the circuit to the pins of the package (bonding). A cover is affixed to the package and the circuits are ready for testing.

In a high-volume production setting this entire sequence of steps is usually carried out in-house. Each of the various phases of the process may be affected by the preceding or following phase. For instance the layout people may request modifications to the circuit in order to get around a particularly difficult routing problem. Such problems may require multiple passes over the offending portions of the design. Close interaction between the specialists is possible and is instrumental in producing a working IC. Unlike those working in the specialized world of the semiconductor industry, a scientist in a research environment must play the roles of computer scientist, circuit engineer, and layout specialist (at the very least). Unfortunately he or she does not have years of

experience or even experienced colleagues to draw upon when problems are encountered. How then can he hope to effectively design VLSI systems?

Two recent developments have been instrumental in bringing IC design into the reach of researchers. Design methodologies, such as described in Mead and Conway [1979], have distilled the complexities of semiconductor devices into a straightforward set of design rules and principles. By encouraging the use of geometrically regular structures and hierarchical design, these structured techniques allow complex IC's to be designed in a relatively short time. The abstraction provided by the rules and principles allow IC's to be designed in a "cookbook" manner, largely ignoring the microscopic behavior of semiconductor devices. The other advance is the availability of standard semiconductor processes, n-channel silicon gate MOS for example, which allow the same circuit to be processed by any of a number of manufacturers with comparable results. Aided by the new design methodologies, a researcher can cope with the complexity of a VLSI circuit; the widely available standard processes assure that his or her circuit can be fabricated.

The priorities of these prototype designers reflect a much different emphasis than those of their counterparts in a mass production environment. A designer working on a system which is intended for marketing is almost surely optimizing the design for some combination of high device density and performance. Conversely, a prototype designer is more interested in trying out a research concept (perhaps an unusual interconnection scheme) and thus will use conservative layout rules and liberal timing margins to insure that the concept and not the technology is the limiting factor. Rapid turnaround (mask generation - wafer fabrication - packaging) is a prime requisite to the research designer since tests on the chips provide important feedback. He or she may try a number of different approaches to

a problem, and count on redesigning the chip several times before reaching a conclusion. In this respect the researcher is not unlike a computer programmer who is implementing a software system: the components of the system may undergo radical changes before the result is blessed, and very little optimization is done until the last iteration, if then. Since these designers are interested in their research and not the technology per se, most prefer to remain ignorant of the details of mask generation and wafer fabrication. Indeed, few understand the detailed physics of semiconductor devices.

Once the designs are completed, the question arises as to who will make the masks, process the wafers, and package the chips. One solution is a complete, in-house facility. In view of the large capital investment required, the skilled personnel support needed, and the prospect that much of the equipment will be obsolete in a few years, most research groups cannot afford this approach. In addition such facilities would not benefit from process improvements common in a larger volume industrial installation. Thus research groups must turn to an outside source, at least for masks and wafer processing. However, it may be practical for a research group to own and operate its own dicing and bonding facility.

The cost of subcontracting mask generation and wafer processing leads to the use of multi-project chips -- IC's which are formed from the juxtaposition of several independent designs. Multi-project chips reduce the cost per idea (or design) tested since it usually does not cost n times the cost for one design to have n designs on a chip. Even so, mask generation for large chips costs about \$7,000 and the fabrication of 10 to 15 three inch wafers costs about \$2,000. Currently a research group might produce one multi-project chip per quarter, composed of approximately 15 projects of several hundred transistors each. Such a chip might be 8 mm on

a side and be further divided by interior scribe lines to facilitate packaging. While one would expect a small percentage yield on a chip of this size, notice that on the average each project takes up only 1/15th of the total area. Therefore it is quite possible for a substantial fraction of the several hundred chips fabricated to contain working versions of most projects.

The realization of rapid mask generation and wafer fabrication turnaround depends on smooth, reliable interaction between the research group, mask house, and fabrication line. An important factor working against smooth interaction is the informal nature of the interfaces that now exist between participants. Mask houses and fabrication lines typically deal on a person-to-person basis with others in the business of manufacturing IC's for sale. Precise documentation of the requirements of each party has been unnecessary, as customers were assumed to have a certain amount of knowledge about the subcontractor's process. The research designer has no such knowledge, worse still, it is often not obvious to either the designer or the subcontractor that an important detail has been overlooked. Misunderstandings about the coordinate system used on a particular pattern generator or the line widths required by a fabrication line can stall the processing of masks or wafers, and ultimately double or triple the time to complete a process which normally requires 3 weeks.

At another level of detail, certain information about a particular phase of the subcontractor's process may speed turnaround or produce a better result. For instance, the manner in which the mask specification data is sorted can significantly reduce mask generation time and cost. Such information is valuable, yet not necessarily available. In the quality conscious semiconductor industry, unusual structures or a feature in an

unexpected place (even if intentional) can cause a delay in processing while the operator verifies the presence or absence of an error. The previous types of problems can be minimized by complete written specification of the requirements of each participant. It is desirable for researchers to be able to treat mask generation and wafer fabrication as "black box" processes, with well defined input requirements and output products. Presently most of this information resides in the heads of various experts at their respective companies; tapping such expertise is a difficult task, but one that will have great benefits.

Researchers' hopes for quick and painless implementation of their IC designs are based on the somewhat naive assumption that a tightly connected sequence of processes can be separated into a series of independent black boxes with immediate success. The simplicity of the goal belies the wealth of detailed knowledge required to effect a solution. Is there really any chance of routinely and quickly implementing prototype VLSI designs? A multi-project chip (9.3mm x 6.3mm, 10 projects) designed at Xerox PARC in the summer of 1978 took 20 weeks for mask generation and wafer fabrication. Largely as a result of that experience, a group at the Massachusetts Institute of Technology were able to have a similar multi-project chip processed in four weeks (thanks to an outstanding effort by all involved parties). The challenge remains to mount a semiconductor industry / research group cooperative effort to implement one-of-a-kind VLSI systems consistently, quickly and inexpensively.

REFERENCE

Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, Reading, Ma., at press.

INNOVATIVE LSI DESIGNS SESSION

Chairperson:

Robert F. Sproull, Assistant Professor of Computer
Science, Carnegie-Mellon University, Visiting
Associate in Computer Science at Caltech

This session displays a cross-section of VLSI designs that comingle logic and memory. One of the strengths of VLSI technologies is the ability to fabricate efficient logic and memory structures with identical fabrication processes on the same substrate. Manufacturers of integrated circuit parts currently exploit this opportunity in only limited ways, such as putting a microprocessor and a small RAM on a single chip. As VLSI circuit densities increase, the advantages of locating logic and memory close together become more pronounced. With decreasing feature sizes, the speed of processing increases faster than does the speed of communication, both on and off the chip. As a result, exploring designs that avoid separating memory and processing is a task of some urgency.

The designs reported in this session couple logic and memory tightly together in configurations designed to address special applications. The application is usually dominated by a particular algorithm that is adapted to use the high concurrency that can be achieved with many logic-memory elements. The algorithm must carefully provide communication among elements, subject to constraints imposed by VLSI fabrication. The solutions vary in several ways: (1) The size of the memory and amount of logic that are welded into a logic-memory element; (2) the number of elements in the system; and (3) the degree of flexibility (or programmability) within the element. The solutions share an abiding concern for regular communication and layout.

Some designs use a large number of elements, each of which contains a fairly small amount of processing and storage. Locanthi describes a way to generate raster-scan display images from descriptions of rectangular regions that make up the image. Each logic-memory element is responsible for saving parameters of a single rectangle and for generating its image. Leiserson shows a design for a priority queue in which each element saves a queue member and communicates with its neighbors in order to modify the queue. The paper of Lea and Sreetharan describes a class of "distributed logic memories" that emphasizes flexibility, so that they may address several applications.

Processing-memory elements have already found widespread application in signal-processing tasks. The paper of Cohen and Tyree describes the role of processing and memory in a synthetic-aperture radar application that calls for about a thousand elements. Here, storage is used primarily for buffering to assure that data arrive at processing stations at the proper time. The need to have data "in the right place at the right time" is a common theme in VLSI designs. The paper by Denny, Buley, and Hatt applies even more elements (about 13,000) to a target-recognition problem; each element has a modest amount of processing, but a prodigious serial memory.

The final paper in the session hints at new uses of processing to provide specialized memory functions that speed the interpretation of high-level languages. The paper by Steele and Sussman presents a design for a machine to interpret the LISP language. It implements storage functions tailored to LISP, helping to interpret data types and to allocate storage (free lists, garbage collection).

VLSI System for SAR Processing

by

Danny Cohen
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

and

Vance Tyree
Caltech/JPL
Pasadena, CA 91125

ABSTRACT

Synthetic Aperture Radar (SAR) is a radar system that processes the return signal to achieve the effect of having a larger aperture than the one provided by the physical dimensions of its antenna. The processing consists of a weighted summation of regularly spaced samples from the signal history, hence of logic for the arithmetic and storage for the signal history. LSI and VLSI technology offer some beautiful ways to implement this computation in chips in which the storage and logic functions are commingled.

The SAR problem discussed in this paper is based on actual requirements set forth by NASA for a spaceborne application.

The requirements for high resolution and high quality necessitate a data sampling rate of 7.5 MHz. For each data value 1,025 4-bit complex multiply+add operations are needed, which is equivalent to 7.7 GHz complex multiply+add operation rate. Since this rate is much too high for general purpose systems, a special-purpose device was sought.

This paper discusses two architectures based on parallel operation of 1,025 identical cells, each of which is capable of performing arithmetic, storage, and several control operations. The operation rate in each device is only 7.5 MHz, which is quite manageable, especially with the help of a substantial degree of pipelining.

A computational-mathematical analysis is used as a primary tool for evaluating the design and some of its tradeoffs.

Two different approaches are discussed and compared; both are based on having 1,025 identical cells working in parallel, but differ in their dual approaches to the flow of data. The mathematics require a relative motion of the data with respect to some (relatively) constant sets of coefficients. In one approach the coefficients are held stationary in space, and the data flows past them; in the other, the data is held and the coefficients flow past.

The paper discusses the architecture, both approaches, some of the control issues, and most important, some aspects of the methodology of the design.

BACKGROUND

Synthetic Aperture Radar (SAR) is a radar system that uses its own motion and information processing capabilities to achieve an effective (virtual) radar aperture which is much larger than the physical aperture provided by its antenna.

Why should the aperture be made larger? Or, what is wrong with the conventional circular-scan radars such as those carried in the noses of most aircraft?

The answer to this obvious question can be found in [1]. It is copied here verbatim:

Experience has shown that most of the images made by circular-scan radar systems aboard airplanes are poorly defined.

The poor definition results from a fundamental reason: Most airborne circular-scan radar antennas are rather small, and fine angular resolution can be obtained only with an imaging system that has a large aperture with respect to the wavelength of radiation received. In other words, the resolution of a large-aperture lens or antenna is finer than the resolution of a small-aperture lens or antenna. The limiting angular resolution is proportional to the ratio between the wavelength received and the size of the aperture.

In conventional optics, the larger the aperture, the higher the quality obtained for certain given conditions. Similarly, in the SAR case, the larger the aperture, the higher the resolution, and the less energy required to obtain a desired signal-to-noise ratio.

The basic idea is very simple. The higher quality image of any ground position is computed from all the radar returns (echoes) from it. The multitude of returns is due to the width of the radar beam and to the motion of the platform relative to the planet. The image obtained by storing these reflections and adding them coherently is better than the one obtained from conventional systems.

The reader interested in the theory and the details of SAR technology is advised to read the article on side-looking radars in Scientific American [1] and the less popular and more detailed literature mentioned in references [2] through [6].

THE MATHEMATICAL PROBLEM

The following is a description of the SAR processing problem, substantially simplified for the purpose of this discussion. The result of the computation is a ground texture map of a swath roughly parallel to and considerably to one side of the track followed by the platform. One simplification employed in this description is to neglect the effects of the altitude of the platform. In the description which follows, think of yourself as looking down on a platform which is moving on a railroad track, the X axis, at velocity v. A map of the area to one side of the track is to be constructed.

At the times $t_i = i\Delta T$, for $i=0,1,2,\dots$ a radar beam is transmitted in the Y-direction. At the times $t_{i,j} = (iN+j)\Delta T$, for $i=0,1,2,\dots$ and for $j=0,1,\dots,N-1$, both the magnitude and the phase of the return are recorded. Let $D_{i,j}$ denote the data recorded at the time $t_{i,j}$.

The set $\{i,*\}$ is called the i th vertical column, and the set $\{*,j\}$ is called the j th horizontal row.

The sampling period, T , is chosen such that the required image pixel spacing along the Y-direction, P_y , is achieved. The relation between T and P_y is $P_y = \frac{1}{2}Tc$, where c is the speed of electromagnetic propagation. The factor of $\frac{1}{2}$ is due to the reflection. Hence the sampling period is $T = \frac{2P_y}{c}$ and the sampling rate is $f = \frac{c}{2P_y}$.

The data $D_{i,j}$ corresponds to the return from the ground position (x,y) .

Conventional radar uses $D_{i,j}$ for $F_{i,j}$, the image corresponding to (x,y) . However, the SAR system computes $F_{i,j}$ by a coherent adding of m returns from each side of (x,y) . Hence

$$F_{i,j} = \sum_{k=-m}^{m} a_{k,j} D_{i-k,j}$$

The number of multiplications required for each point $F_{i,j}$ is $2m+1$. Since a new value of $D_{i,j}$ is recorded every time period T , the multiplication rate required is $r = (2m+1)/T = (2m+1)f$.

It is worth mentioning again that this description is an extreme simplification of the real problem. Among the factors omitted for the sake of simplicity and clarity are the effects of the angle between the planet motion and the platform velocity and the distance variation of each surface position from the system as a function of j and k .

In addition, the system is described here as if the platform is at ground level (whereas 800 Km is a typical altitude), as if the $\{t_{i,j}\}$ are uniformly distributed (whereas typically there is an inter-beam waiting period for range gating etc.), and as if the pixel spacing is based only on the range (rather than the slant-range).

These simplifications are made here since they do not change the basic concepts of the system. As a matter of fact, the system which is now being VLSI-implemented without benefit of these simplifications is very similar to the one described in this paper.

The relation between the pixel-spacing and the system resolution depends on many factors (e.g., the value of m). The exact relation is also left out of this paper. It is sufficient, for the purpose of this paper, to assume that the resolution is close to the pixel spacing.

TYPICAL NUMBERS

The following numbers used as examples are taken from the requirements for SEASAT-A, which may be found in reference [10].

$$\begin{array}{ll} p_y = 20 \text{ meters} & N = 1,024 \\ f = 7.5 \text{ MHz} & m = 512 \text{ points on each side} \end{array}$$

This implies a multiplication rate of $r = (2m+1)f = 1,025 \times 7.5 \text{ MHz} = 7.7 \text{ GHz}$

First, the bad news about these numbers. Since both the coefficients $\{a_{i,j}\}$ and data $\{D_{i,j}\}$ are complex quantities, each multiplication requires 4 real multiplications. Therefore, the rate of real multiplications is about 30.75 GHz.

The good news is that both the coefficients and the data are handled with only 4 bits of significance. However, the accumulation is performed in 18-bit complex arithmetics.

DISCUSSION

Let Z be the operator which delays data by the time period T . However, Z does not affect the constant coefficients.

Hence $Z D_{i,j} = D_{i,j-1}$ for $j > 0$, and $Z D_{i,0} = D_{i-1,N-1}$

Similarly $Z^N D_{i,j} = D_{i-1,j}$, but $Z b_{i,j} = b_{i,j}$, where the b 's are constant coefficients.

By definition, we have

$$F_{i,j} = \sum_{k=-m}^{m} a_{k,j} D_{i-k,j} = \sum_{k=-m}^{m} a_{k,j} Z^{kN} D_{i,j}$$

This is mathematically (i.e., formally) correct. However, it is unrealizable, since it involves negative powers of the operator Z . Since Z means delay, the operator Z^{-1} means prediction. Since we do not know how to build the prediction operator, it must be circumvented.

F can also be represented by

$$z^{mN} F_{i,j} = \sum_{k=-m}^m a_{k,j} z^{(m+k)N} D_{i,j}$$

Define $h \triangleq k+m$, and obtain

$$z^{mN} F_{i,j} = \sum_{h=0}^{2m} a_{h-m,j} z^{hN} D_{i,j}$$

Define $b_{k,j} \triangleq a_{k-m,j}$ and obtain

$$z^{mN} F_{i,j} = \sum_{k=0}^{2m} b_{k,j} z^{kN} D_{i,j}$$

This means that by implementing the operations shown on the right-hand side of the above equation, the image function F obtained is delayed by mNT . This is not surprising, since the definition of F requires m neighboring data values on each side. Hence, let $G \triangleq z^{mN} F$ represent the delayed image function.

APPROACH (A)

The implementation of $G = \sum b_{k,j} z^{kN_D}$ is discussed and analyzed in [7]. It is shown there that an optimal implementation (with respect to a set of design objectives defined in that reference) is realized from the following presentation

$$G_{i,j} = \sum_{k=0}^{2m} z^k b_{k,j} z^{k(N-1)} D_{i,j}$$

This computation can be implemented by the circuits consisting of $2m+1=1,025$ cells as shown in figure 1. Please remember: the z^k does not affect the b 's.

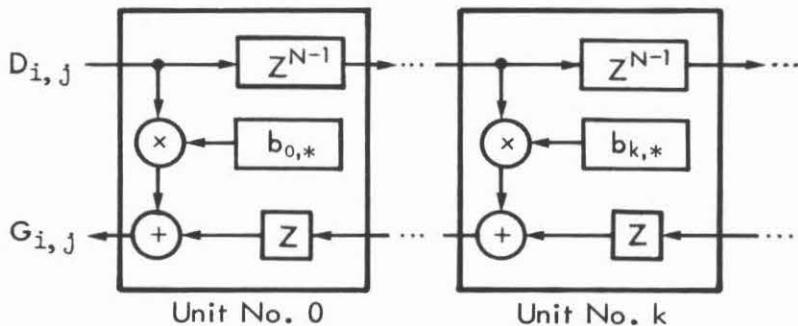


Figure 1: The cells for approach (A).

This implementation is systolic¹ and is relatively easy to implement in VLSI. The complex multiplication rate of each cell is only

$$r' = r/1,025 = 7.5 \text{ MHz}$$

¹H.T. Kung and C.E. Leiserson, in [8], write, "A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network."

which for only 4-bit real terms is well within the performance range of off-the-shelf, commercially available LSI multipliers [9].

It is possible to use pipeline multipliers because the data can easily be arranged such that it is available serially, with the least significant bits leading, and because the delay introduced by the pipeline is insignificant.

This approach allows the application of even slower circuits, which may be beneficial for power and size considerations.

Figure 2 shows how the coefficients, $\{b_{k,j}\}$, can be stored in sequential memory (circular shift registers) rather than in random access memory. This also may be beneficial for power and size considerations.

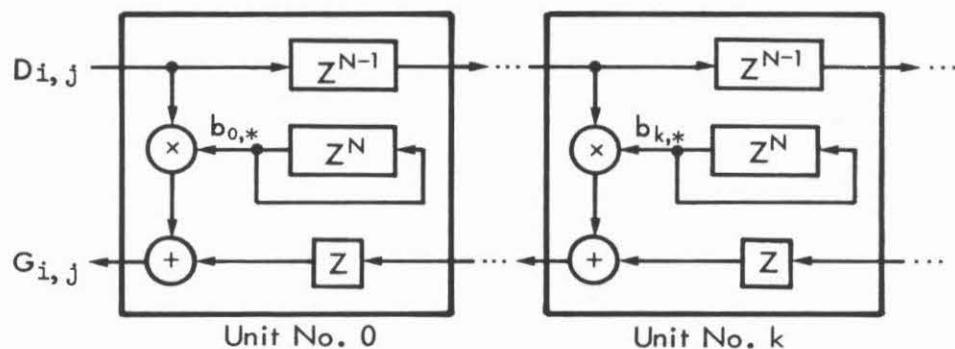


Figure 2: Approach (A), with circular shift registers for the b 's.

In summary, $2m+1$ cells, arranged in a linear sequence, are used. The k th cell (for $k=0, 1, 2, \dots, m$) computes the contribution of the $(m-k)$ th column ahead/ago, to the image of the current position column. This cell stores $2N$ complex data quantities: N coefficients, $\{b_{k,*}\}$, $N-1$ input values, and 1 partial sum.

Each device performs one complex multiplication and one complex addition at the sampling rate, f .

APPROACH (B)

This approach is dual, in a way, to the previous one. Whereas in (A) each cell computes a single selected phase of all the image columns $\{G_{i,*}\}$, in (B) each cell computes all the phases of selected image columns only. As before, there are $2m+1$ devices, numbered $k=0,1,\dots,2m$.

From now on all the first indices (such as the "i" in $b_{i,j}$) are computed in modulo $(2m+1)$ arithmetic.

In this approach the k th cell computes the image columns $G_{i,*}$ for all values of i , such that $i \equiv k \pmod{2m+1}$.

Hence, if the entire image is considered as a series of "frames", each composed of $2m+1$ vertical image columns, then each cell produces all the images which belong to a certain column in all the frames.

The basic module of the system, according to this approach, is an accumulator as shown in figure 3.

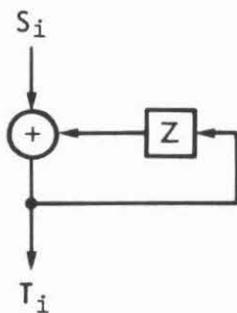


Figure 3: The basic accumulator.

It is easy to see that

$$T_i = S_i + T_{i-1} = S_i + Z T_i$$

$$T_i - Z T_i = (I-Z) T_i = S_i$$

$$T_i = (I-Z)^{-1} S_i = \sum_{k=0}^{\infty} Z^k S_i$$

This holds, obviously, if $Z^k=0$ for some $k>0$.

This shows, not very surprisingly, that each T_i is the sum of some of the previous input values $\{S_j\}$ for $k < j \leq i$.

If the input sequence is $\{S_{i,j}\}$ for $i=0,1,\dots$ and $j=0,1,\dots(N-1)$, and if the Z is replaced by Z^N (a shift register of length N), then

$$T_{i,j} = \sum_{k=0}^{\infty} Z^{kN} S_{i,j}$$

Hence, the column $T_{i,*}$ is the sum of some of the previous columns.

In order to use this accumulator for the SAR processor the $\{S_{i,j}\}$ should be the products of the input values $\{D_{i,j}\}$ by the coefficients, and the summation should include only $2m+1$ terms.

In order to limit the range of the summation the cell is modified such that the input to the accumulators-column, Z^N , is cleared at all times (i,j) whenever $i \equiv k \pmod{2m+1}$. Hence, this occurs for N successive cycles, every $N(2m+1)$ cycles, or for one column-period every frame-period.

The modified cell is shown in figure 4. Let $G_{i,j}^{(k)}$ be the output of the adder at $t_{i,j}$, in which the raw data $D_{i,j}$ is multiplied by the coefficient $b_{u,v}$, for some yet undetermined values of u and v .

One can verify that

$$G_{i,j}^{(k)} = \sum_{h=0}^{\beta} b_{u,v} D_{i-h,j} \text{ where } \beta \equiv i-k-1 \pmod{2m+1} \text{ and } 0 \leq \beta \leq 2m$$

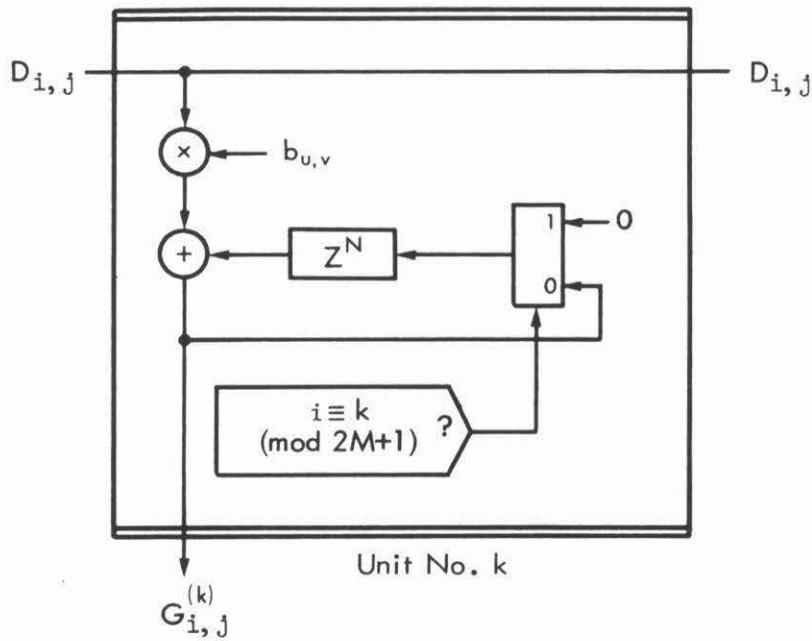


Figure 4: The modified cell, for the k th phase.

Hence, at the time $t_{i,j}$ the $G_{i,j}^{(k)}$ is the weighted sum of the previous $i-k \pmod{2m+1}$ columns. In order to have $G_{i,j}^{(k)} = G_{i,j}$, we consider it only when $\beta=2m$, i.e., when $i \equiv k \pmod{2m+1}$, and get

$$G_{i,j}^{(k)} = \sum_{h=0}^{2m} b_{u,v} D_{i-h,j}$$

If u and v are chosen to be $u=k-i+h \pmod{2m+1}$ and $v=j$, then

$$G_{i,j}^{(k)} = \sum_{h=0}^{2m} b_{h,j} D_{k-h,j} = G_{i,j} \quad \text{when } i \equiv k \pmod{2m+1}$$

We have

$$\begin{aligned} G_{i,j}^{(k)} &= \sum_{h=0}^{\beta} b_{k-i+h,j} D_{i-h,j} = \\ &= \sum_{h=0}^{\beta} b_{k-i+h,j} z^{hN} D_{i,j} = \sum_{h=0}^{\beta} z^{hN} b_{k-i+h,j} D_{i,j} \end{aligned}$$

Introduce Z_b , the delay operator which operates only on the coefficients (the b 's) similarly to the operator Z , which operates only on data

$$G_{i,j}^{(k)} = \sum_{h=0}^{\beta} z^{hN} Z_b^{-(k+h)N} b_{-i,j} D_{i,j} = Z_b^{-kN} \sum_{h=0}^{\beta} z^{hN} Z_b^{-hN} b_{-i,j} D_{i,j}$$

This expression requires some interpretation: since the $\{b_{i,j}\}$ are known coefficients, negative powers of the delay are allowed. Since the first index is computed modulo $2m+1$, a single circular shift register of length $N(2m+1)$ can be used to store all the $\{b_{i,j}\}$.

Since the "nominal" b in the above expression is $b_{-i,j}$ the $\{b_{i,j}\}$ are arranged such that each $b_{i,j}$ is followed by $b_{i,j+1}$ (like the data, $D_{i,j}$), but the entire column $b_{i,*}$ is followed by the column $b_{i-1,*}$ (unlike the data).

Since the k th cell requires the phase $-kN$ (as suggested by the term in front of the \sum -sign), each cell taps the circular shift register N units apart. Hence each cell contains a shift register of length N for the data (Z^N) and a shift register, also of length N , for the coefficients (Z_b^N).

However, only when $i \equiv k \pmod{2m+1}$ is the output $G_{i,j}^{(k)}$ a valid value for the required $G_{i,j}$. Therefore, the k th cell is allowed to announce its output only then. This permission-to-announce is implemented with a tri-state driver. The k th device therefore has the structure shown in figure 5.

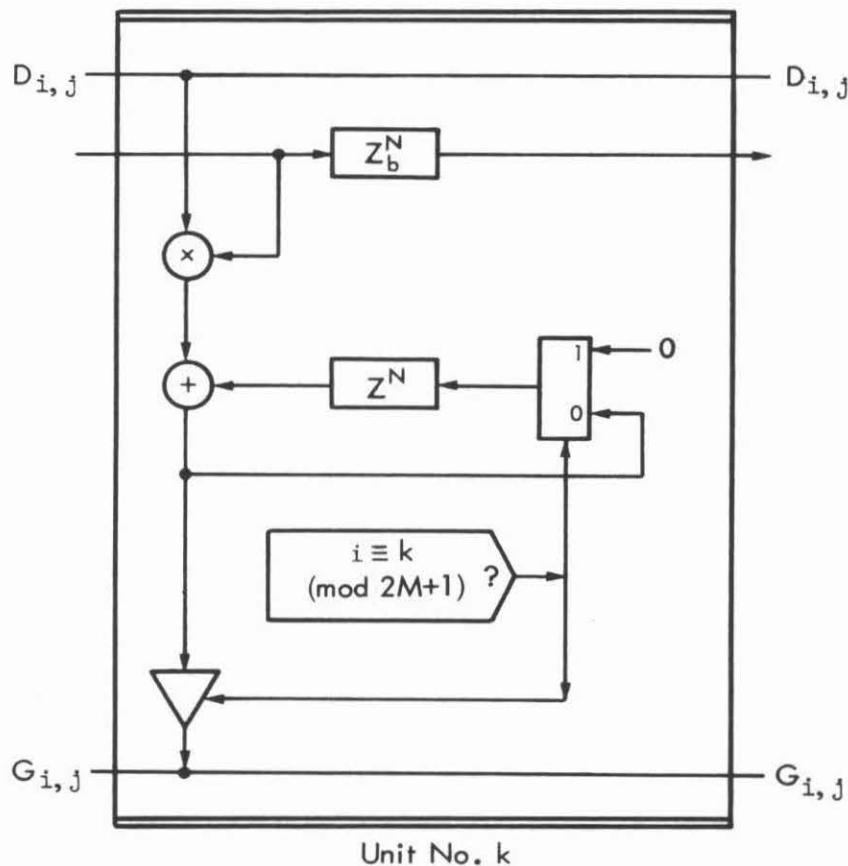


Figure 5: The k th cell for approach (B).

In summary, $2m+1$ cells, arranged in a linear sequence, are used. The k th cell (for $k=0,1,2\dots 2m$) performs all the computation for all the image columns $G_{i,*}$ such that $i \equiv k \pmod{2m+1}$. This cell stores $2N$ complex quantities: N coefficients, $\{b_{i,j}\}$, and N partial sums.

Each cell performs one complex multiplication and one complex addition at the sampling rate, f .

The linear sequence arrangement is needed only for the coefficients. However, any other arrangement (e.g., tree) can be used for the input data $\{D_{i,j}\}$ and the output image, $\{G_{i,j}\}$.

ABOUT THE IMAGE PIXEL SPACING

The horizontal pixel spacing, P_x , of the system is $P_x = NTV$, whereas the vertical pixel spacing, P_y , is $P_y = \frac{1}{2}T_c$. The ratio, R , between these pixel spacings is

$$R = \frac{P_y}{P_x} = \frac{T_c}{2NTv} = \frac{c}{2Nv}.$$

For the numbers used here, and for $v = 36,000 \text{ Km/h} = 10^4 \text{ m/sec}$ (which is about 22,500 mph) we get

$$R = \frac{c}{2Nv} = \frac{300,000,000}{2 \times 1,025 \times 10,000} \approx 15$$

This suggests that it is possible to reduce the output in the X-direction by a factor P (which does not exceed R) without degrading the image quality.

Obviously, it is desirable to take advantage of the reduced requirement for output, and to reduce the computation accordingly.

This advantage can be achieved by computing only every P th image column, $\{G_{i,*}\}$, e.g., for $i \equiv 0 \pmod P$.

COMPARISON OF THE TWO APPROACHES

The single most important issue of the architecture of the SAR processor is the dynamics of the data flow (shall we call it data-dynamics?). It is clear that the data and the coefficients must flow past each other.

The first approach keeps all the coefficients for each image-column, $\{b_{k,*}\}$, always in the same device and keeps all the data flowing past them. Hence, relatively stationary coefficients with dynamic data.

The second approach keeps all the data required for computing a certain image-column in the same device and keeps all the coefficients flowing past them. Hence, relatively stationary data with dynamic coefficients.

Approach (A) is completely laminar flow systolic system, composed of $2m+1$ devices, each of which is systolic, too. Approach (B) is a periodic-laminar flow systolic system, composed also of $2m+1$ devices which are systolic only periodically.

The operation control is not discussed here for either approach. In both cases it is simple and straightforward if both broadcast and daisy-chained connections are used. The problem of assigning an individual identity (e.g., the value of k) to each device, dynamically, in spite of identical hardware can be solved in any of several ways, for either approach.

Generally, the operation control is similar (in complexity, connectivity etc.) for both approaches.

Dynamic data allows simpler access to the resulting $G_{i,j}$, because it is always produced by the same device. One may notice that approach (A) does not require the tri-state output drivers which (B) requires. This is an advantage of (A) over (B).

Dynamic coefficients, as in approach (B), significantly simplify the process of changing the coefficients, $\{b_{i,j}\}$, (when needed due to changes of the flight parameters) because all the coefficients circulate through a single

loop, which allows for easy external injection of the new set of coefficients. This is not just a simplification, but also a simple way to achieve a synchronous and an instantaneous change. Therefore, this is a significant advantage of (B) over (A).

In (A) an arithmetic malfunctioning (multiplication or addition) in a single cell, affects all the image points, whereas in (B) it affects only the columns computed by this particular cell. Hence, (B) is much more robust than (A).

Suppose that $\{G_{i,j}\}$ is to be computed only for a certain subset of columns, for example only for one column in P (i.e., $i=nP$). How does this affect the architecture? In approach (A) no saving of processing hardware can be achieved without major modifications of the architecture, whereas in (B) only the devices corresponding to these columns have to be implemented. However, the Z_b^N of each eliminated cell should be included in the system such that a total $Z_b^{N(2m+1)}$ circular shift register is maintained.

The last three considerations are overwhelming reasons to prefer approach (B) over (A). Therefore, the system which is currently being implemented in VLSI is based on (B).

VLSI IMPLEMENTATION

The control signalling of the system can be based on the application of two regimes of communication in the system: one is a "hop-by-hop" communication, like a daisy chain, between the cells, and the other is based on simultaneous broadcasting to all the units.

In order to reduce drastically the total power consumption of the system, a technology is being developed to allow these devices (each being a VLSI chip, e.g., a "lot" of silicon), to be interconnected without having to cut the wafer. This necessitates checking each device on the wafer. Printing metal connections on the wafer makes it possible to connect the good ones and skip the bad ones.

There are many other important details: the organization of a linear array on a round wafer, loading the coefficients into different memories (for the former approach, only), dynamically assigning identification to the devices, increasing system robustness, and so forth.

These details are very important, and have conceptually simple solutions. We did not find it necessary to include them here.

Jet Propulsion Laboratory was funded in FY77 to develop and demonstrate real-time SAR processor technology that would enable on-board spacecraft SAR processing. A custom VLSI implementation of approach (B) described in this paper will be an important factor in enabling an on-board SAR processor. This VLSI device, the Azimuth Correlator Device (ACD), is being designed and fabricated at TRW, Inc., and contains all of the functional elements of approach (B) (figure 5) with some additional circuitry to perform control functions and to correct far migration of image elements through the processed aperture.

The range migration results from a simple geometric relationship between the SAR instrument and an image element on the surface of the earth (or other planet) that results in a change in surface element range as the surface element occupies different azimuth positions within the area illuminated by the radar antenna.

A functional block diagram of the ACD (figure 6) shows the Range Migration Compensation (RMC), which is divided into coarse and fine components (RMC-C and RMC-F). The Azimuth Reference Function (ARF) coefficients and the RMC coefficients are stored in shift register memory, which also serves as the delay operator in the ACD. The ACD contains a complex multiplier, a complex adder, shift register memory to store coefficients and partially processed image elements, and control and timing circuitry.

A more detailed description of this VLSI device is contained in [11]. It is expected that the first lot of ACD chips will be available for testing in the third quarter of 1979.

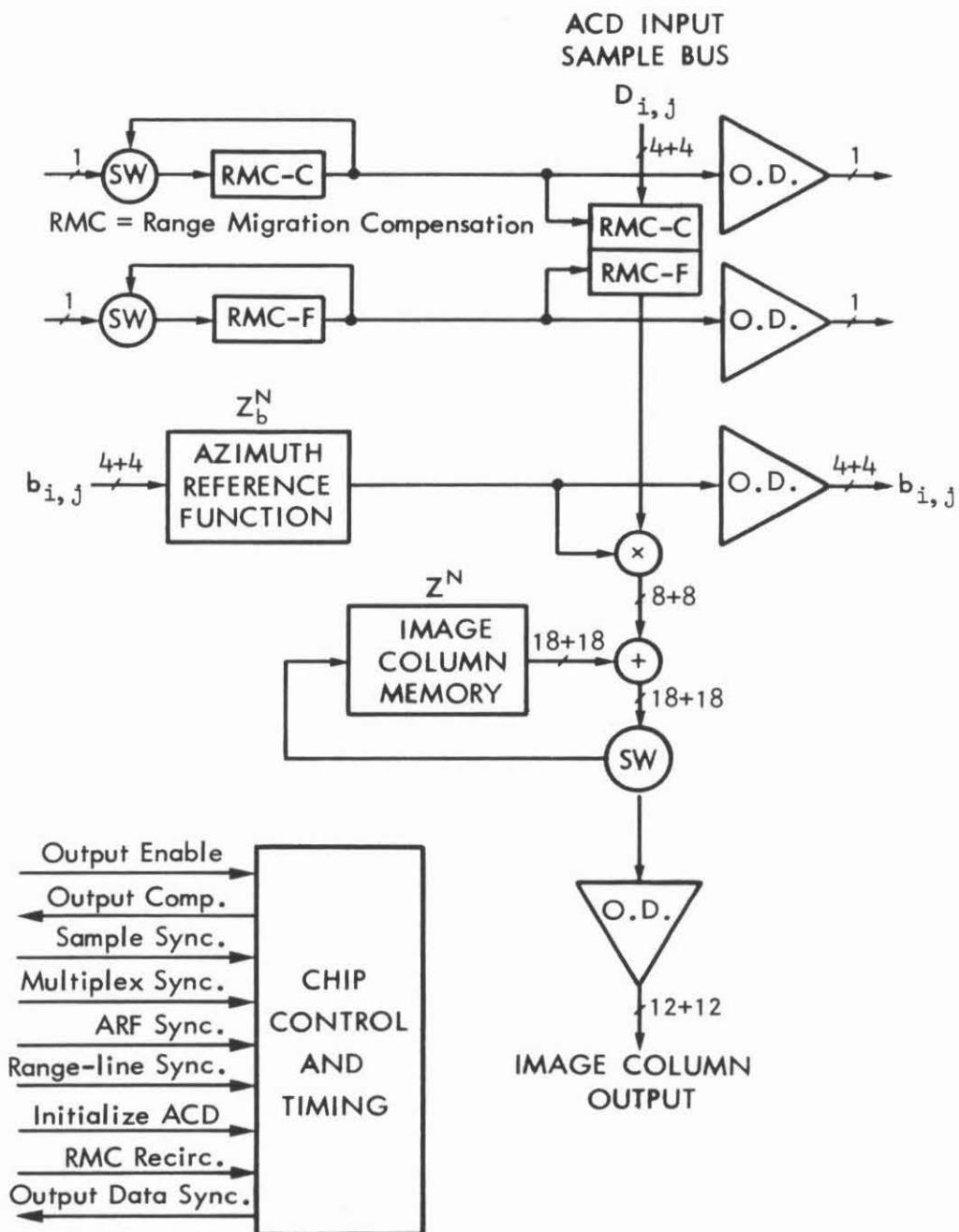


Figure 6: A block diagram of the Azimuth Correlator Device.

CONCLUSION

SAR processing requires a high rate of arithmetic operations and a substantial amount of data storage.

It is possible to implement a SAR processing system based on a multitude of identical cells, all working in parallel, at full capacity, such that they share uniformly the required system operating rate.

In addition, it is possible to organize such a system with only a small number of interconnections.

Therefore, VLSI is most suitable for SAR processing.

ACKNOWLEDGMENTS

The authors would like to extend their appreciation to C. Wu, R.G. Pierson, W.E. Arens and R. Lipes for their contributions to the development of the SAR processor studies that lead to the conception of Approaches (A) and (B). The generalized mathematical representation of a SAR azimuth processor described in this paper was an outgrowth of these earlier studies.

BIBLIOGRAPHY

- [1] Jensen, H., Graham, L.C., Porcello, L.J. and Leith, E.N., "Side-looking Airborne Radar", Scientific American, October 1977, 84-95.
- [2] Cutrona L.J., Leith, E.N., Porcello, L.J. and Vivian, W.E., "On the Application of Coherent Optical Processing Techniques to SAR", in Proceedings of the IEEE, Vol. 54, No. 8, August 1966, 1026-1032.
- [3] Brown, W.M. and Porcello, L.J., "An Introduction to Synthetic Aperture Radar", IEEE Spectrum, Vol. 6, No. 9, September 1969, 52-62.
- [4] Synthetic Aperture Radar, ed. John Kovaly, Artech House, 1976.
- [5] Synthetic Aperture Radars, Theory and Design, by Robert O. Harger, Academic Press, 1970.
- [6] Proceedings of the Synthetic Aperture Radar Technology Conference, March 8-10, 1978, Las Cruces, New Mexico. This issue can be obtained from: SARTC Publications Committee, Physical Sciences Laboratory, Box 3-PSL, Las Cruces, New Mexico, 88003.
- [7] Cohen D., "Mathematical Approach to Iterative Computation Networks", Proceedings of the 4th Symposium on Computer Arithmetics, October 1978, Santa Monica, California, pp. 226-238, IEEE Catalog No. 78CH1412-6C. Also available as USC/Information Sciences Institute Research Report RR-78-73.
- [8] Kung, H.T. and Leiserson, C.E., "Systolic Arrays (for VLSI)" to appear in Introduction to VLSI Systems by C.A. Mead and L.A. Conway, Addison-Wesley, 1979.
- [9] Data sheets for the LSI multipliers MPY-16A, MPY-16AJ and TDC1010J, TRW LSI products, Redondo Beach, California, 1978.
- [10] Rolando, J., "The SEASAT-A Synthetic Aperture Radar Design and Implementation", Proceedings of the Synthetic Aperture Radar Technology Conference, March 8-10, 1978, Las Cruces, New Mexico.
- [11] Tyree, V., "Custom Large Scale Integration Circuit for Spaceborne SAR Processors", Proceeding of the Synthetic Aperture Radar Technology Conference, March 8-10, 1978, Las Cruces, New Mexico.

LOGIC-ENHANCED MEMORIES: AN OVERVIEW AND SOME EXAMPLES
OF THEIR APPLICATION TO A RADAR TRACKING PROBLEM

By

W. Michael Denny

Ernest R. Buley

Earl Hatt

General Research Corporation

Santa Barbara, California

1 INTRODUCTION

For the past two years, General Research Corporation has been investigating ways of using very large amounts of active memory to solve some stressing data processing problems encountered in ballistic missile defense systems. Our approach has been to use large numbers of simple logic elements (from 2 thousand to 2 million elements, each of some 200 gates) distributed throughout very large memories (10^{12} bits).

Two aspects of this work give it a unique place in the current milieu of research on von Neumann architectures. First, the work has been directed toward fairly specialized problems in ballistic missile defense. This problem-directed approach has been consciously adopted in view of the relatively undeveloped state of highly parallel algorithms and architectures. We have found that the exigencies of the problem itself guided us through three separate architectural designs to solve one problem. These different designs reflected different algorithmic approaches to the problem and would be implemented by different circuit technologies. (We are now constructing a portion of one design, of sufficient size to permit a more detailed proof-of-principle verification.) Until the algorithmic and architectural theory of parallel structures matures, we expect much practical design to be intuitively guided. Perhaps the experiences reported here will help guide the intuitions of others.

Second, the highly parallel structures we describe are basically memory structures. The ratio of logic gates to memory bits is roughly 10^{-5} . However, in these structures, memory is not treated as a passive functional unit characterized only by storage capacity and access time. Rather, the memory is seen as an active component which cooperates with a more conventional processor in the data processing task. Although the logic-enhanced memory performs much of the processing, we have not insisted that it be responsible for all of it. Much work remains to be done in exploring the appropriate division of labor between the "processor" and its logic-enhanced memory.

1.1 ARCHITECTURAL CONTEXT

Traditional memories have been considered relatively passive functional elements in data processing. They are characterized by their reliability, size, and speed and by the way their contents can be retrieved (referenced) by the processor. Memories which transform the stored data themselves are rare. Although recently memories have gradually taken a more active role, their increased internal activity has generally been in the interests of increased reliability (error detecting/correcting memories), increased storage and retrieval speed (buffer and cache memories), and increased size (hardware "paging boxes" and other hardware-managed virtual memory hierarchies).

These low levels of active memory architectures have themselves spawned a more complex category of active memories. As the size and effective speed of memories increases, processors spend more and more time threading their way through the disorganized and arbitrarily organized data in order to find the data needed at various processing steps. Memories whose internal activity assists the processor in this chore represent a first step toward a more equitable division of the processing load between the processor and storage functions. "Defined field"¹ memories remove much of the arbitrary data organization imposed by the memory's word length. Content Addressable Memories (CAMs) remove the artificial organization imposed on the data by its physical location in memory by allowing the contents of the data itself to express some simple aspects of its organization. FIFO and LIFO memory organizations implement in hardware a few of the more common data structures which asynchronous processes, expression evaluation, block structured languages, and structured control have found useful.²

1.2 THE LOGIC-ENHANCED MEMORY

Architectural considerations such as those above have been partially responsible for the direction of our recent research. We are exploring a class of active memory structures which we believe represents another step in the direction of more equitable distribution of labor between the processing and storage elements. One approach to such processor/memory combinations is a simple extension of Content Addressable Memories (CAMs).

Sometimes the simple match/don't-care or bounded search criteria used by CAMs to associate data are not appropriate. We might like to know which data are related by simple functional transformations or satisfy more general relationships than match/don't-care or bounded search. For example, we might ask which sets of data not only are identified by a match/don't-care condition, but also satisfy the criterion that each value-object in the set, when divided by a corresponding value-object in some other set, produces a result approximately equal to some value. The parameters specifying "approximately equal" could be passed to this type of memory along with the value the division should produce for a "hit" to be recorded. Such a memory would make an effective matched filter in signal processing applications. And analogous to the highly parallel match circuits found in a CAM, this example of a "logic-enhanced memory" would have a highly parallel set of divide and compare

¹W. T. Wilner, "The Design of the Burroughs B1700," FJCC, 1972; and "Memory Utilization on the B1700," FJCC, 1972.

²Yaoh-an Chu, High Level Language Computer Architecture, Academic Press, 1975, pp. 63-107.

circuits to transform the data and determine which ones should be retrieved. While not reported here, such a logic-enhanced memory has been designed at General Research Corporation in a separate effort. Its internal structure is very much like that of the third solution reported in Sec. 2 of this paper.

Simple and highly parallel transformations applied to the data can be used not only to aid retrieval, but to enable a memory to produce its own data for subsequent processing. Finite difference methods fit nicely with the concept of arithmetic simplicity of the data transformation done by the logic-enhanced memory. An example of a bistatic radar "target finder" using just such an approach is given in Sec. 2 of this paper.

We have found that when traditional processor-memory architectures are used along with such logic-enhanced memories, their computational burden is reduced by several orders of magnitude. For example, the bistatic target finding problem discussed in Sec. 2 can be shown to require a computational power equivalent to about twenty PEPE-class processors. However, a logic-enhanced memory architecture resembling an array of 12,288 Babbage-like differential engines distributed over 6×10^{11} bits of storage reduces the magnitude of the problem to one well within the capacity of a modern minicomputer or microprocessor.

Logic-enhanced memories also promise a mechanism for utilizing very large memories (1 to 10 billion bytes). Because the LEM does much of the manipulation of large amounts of data, and presents the processor with much smaller quantities of data (LEMs give the processor only the "right" data), the processor's address space can be much smaller than that actually spanned by the LEM memory. This is of most value for processors in the mini-micro range and suggests that LEMs and arrays of microprocessors or small minicomputers could be profitably combined. Microprocessor arrays promise relatively large amounts of processor power over a small (< 24 bits) address space. A LEM architecture could provide part or all of the storage used by an array of microprocessors.

In the LEM architectures discussed in Sec. 2, it is important to realize that although some 10^{12} bits of storage are used, the minicomputer using the LEM does not directly address any of it except the "top" containing the proper data. In this sense the LEM's role in a problem is analogous to the role played by a stack in expression evaluation: both allow large amounts of storage to be used, but not directly addressed by the processor. This gives rise to the tantalizing notion that LEM architectures might not only reduce the address space requirement of some conventional processors, but under some circumstances, eliminate the notion of addressable storage altogether--and with it the host of address-related problems inflicted on computer architecture.¹

¹ See, for example, W. Lonergan and P. King, "The Design of the B5000," Datamation, Vol. 7, No. 5, pp. 28-32, May, 1961; and P. Christy, "Mini-computer Architecture Links Past and Future Generations," Electronics, July 6, 1978, pp. 98-105.

Furthermore, the logical complexity of the functional elements which we have thus far used in our LEM designs (100-500 gates) is far below that of current microprocessors (4000-7000 gates). (This low level of complexity suggests that, while a microprocessor implementation of a LEM might be a worthwhile research tool, it would be inferior to a custom LSI or VLSI implementation in size-critical applications.) The LEM logic elements not only have one to two orders of magnitude fewer gates than microprocessors, but are an order of magnitude faster. This leads one to consider the LEM elements and microprocessors as forming levels in a hierarchy. The LEM elements are valuable in data-intensive problems, just as ALUs are valuable components in computation-intensive algorithms. Computationally stressing calculations can themselves often be traded off for more memory. With the advent of high-density memories, data-intensive algorithms using some simple, highly-distributed logic to process the data appear to be cost-effective alternatives in problems traditionally requiring very fast processors.

To date, we have developed one LEM design in enough detail to permit prototype manufacture. This design, described in Sec. 2, produces data at a rate that a processor in the PDP-11/45 class can handle. It remains to be seen whether subsequent LEM designs have similar characteristics and whether they are appropriately matched to microprocessor arrays.

Another feature of the simplicity of the logic elements in LEMs is the possibility of tight coupling of the algorithm embodied in the logic to the physical properties of the storage medium itself. At first glance, this philosophy may seem contrary to the trend in hardware/software design toward increasing isolation of the algorithm from the hardware. However, even though the algorithm implemented in the LEM may itself be very hardware-dependent, its effect is very often to isolate the other processing elements from the intricacies of the storage unit's physical organization. The "difference engine" LEM design in Sec. 2 is an illustration of this effect. As different types of novel storage mechanisms proliferate, LEM designs can serve very effectively in allowing the processor to treat storage in an algorithmically structured manner independent of the physical organization of the storage devices. Furthermore, because the LEM logic is itself relatively simple, the fact that it is tuned to the hardware in which it is implemented is not so distasteful as it is with more complex algorithms running on "general purpose" processors.

A final consequence of LEM logic's simplicity is more of a hope than an observed property. If the logic itself needs only 100 gates or so, it is more likely to be implementable in the same technology as the memory itself--probably even on the same chips. The advent of custom VLSI makes this possibility a very economically attractive one.

2 THREE LOGIC-ENHANCED MEMORY DESIGNS

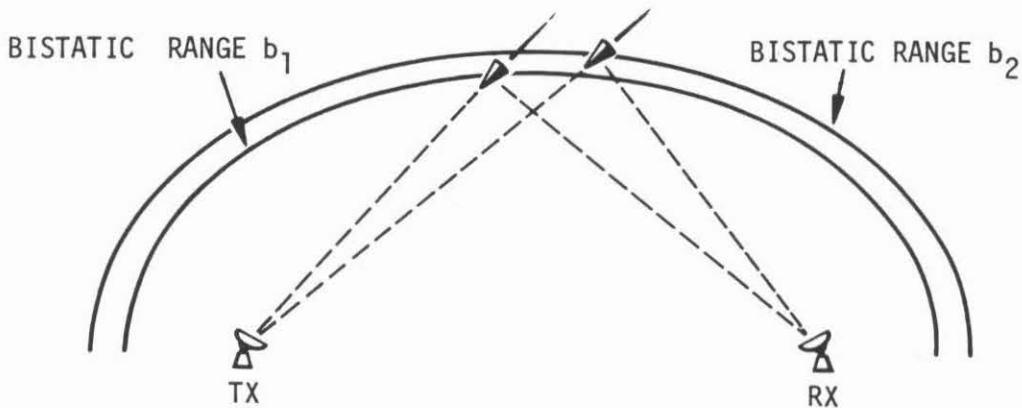
2.1 THE "DEGHOSTING" PROBLEM

One type of radar system that has been considered for ballistic missile defense is a "bistatic multilateration" target locator. Such a system works like this.

Suppose we have a pair of radars--one which transmits and one which receives the pulse reflected from a target, as illustrated in Fig. 1. The only information that the receiving radar extracts from the reflected pulse is the time since it was transmitted. This information is enough to locate the target on an ellipsoid whose foci are the transmitter and receiver and which is rotated about a line joining these foci. In two dimensions, the "bistatic range"--from transmitter to target to receiver--defines an ellipse, and it takes one more receiver to locate the target at the intersection of two ellipses.

Serious problems begin to emerge if there are several targets in the field of view, as in Fig. 2, where there are four intersections of the ellipses defined by the bistatic ranges of two receivers.

In two dimensions, a third receiver is necessary to separate the "real" intersections from the "ghost" intersections. The third receiver



AN-49203

Figure 1. A bistatic range measurement locates an object on the surface of an ellipsoid of revolution.

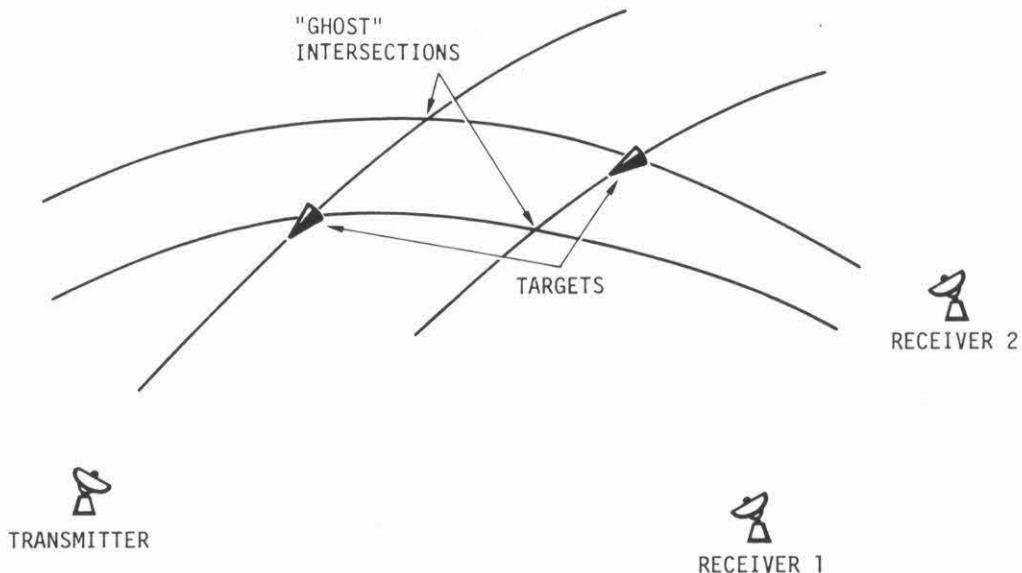


Figure 2. Multiple targets produce "ghost" intersections

defines an additional ellipse for each target. Targets are then recognized by the intersection of all three ellipses, whereas "ghosts" are located at intersections of only two ellipses.

In three dimensions, four receivers are required to locate multiple targets. And, in practice, because of noise in the bistatic transmit times and the vagaries of radar transmission and reception, a 5-out-of-6 scheme would be used. It can be shown that the number of "ghost" intersections is of the order of N^3 , where N is the number of targets in the common viewing volume of the single transmitter and several receivers.

Forty targets, a realistic threat density, would result in 64,000 "ghosts". The real-time requirements of the defense system dictate that the N real targets be separated from the N^3 ghosts in about 3 ms--imposing an enormous load on a sequential processor. Forty targets would (conservatively) require about 20 PEPE-class processors. This large amount of required processing, coupled with the inherent parallelism of the problem, prompted us to examine the logic-enhanced memory as a possible solution.

2.2 SOLUTION ONE

The past thirty years of computing have given us some intuition about problem-solving on sequential machines, but very little about problem-solving with highly parallel structures. In our approach to this problem we found it advantageous to imagine the volume around the radars as being subdivided into "elemental volumes" in various ways. Mentally, we could picture each such elemental volume of space as associated with a LEM processing element--a simple logic-memory combination. We then imagined each elemental volume being processed in parallel with all the rest. (A side effect of viewing the problem in such spatial terms was that if each LEM element could do its work independent of the number of targets, then the processing time for the whole group of LEM elements could also be nearly independent of the number of targets. This desirable property is in sharp contrast to a sequential processor, where the processing time for the classical algorithm rises exponentially with the number of targets.)

Our first approach to this problem was to imagine the entire volume around the radars divided into cubes (see Fig. 3), each associated with a small processing element. Each element is pre-loaded with the values of bistatic range that determine an ellipsoid passing through its cube. Each element then reports only when five (of the possible six) ellipsoids intersect in its cube of space. A logic-enhanced memory built

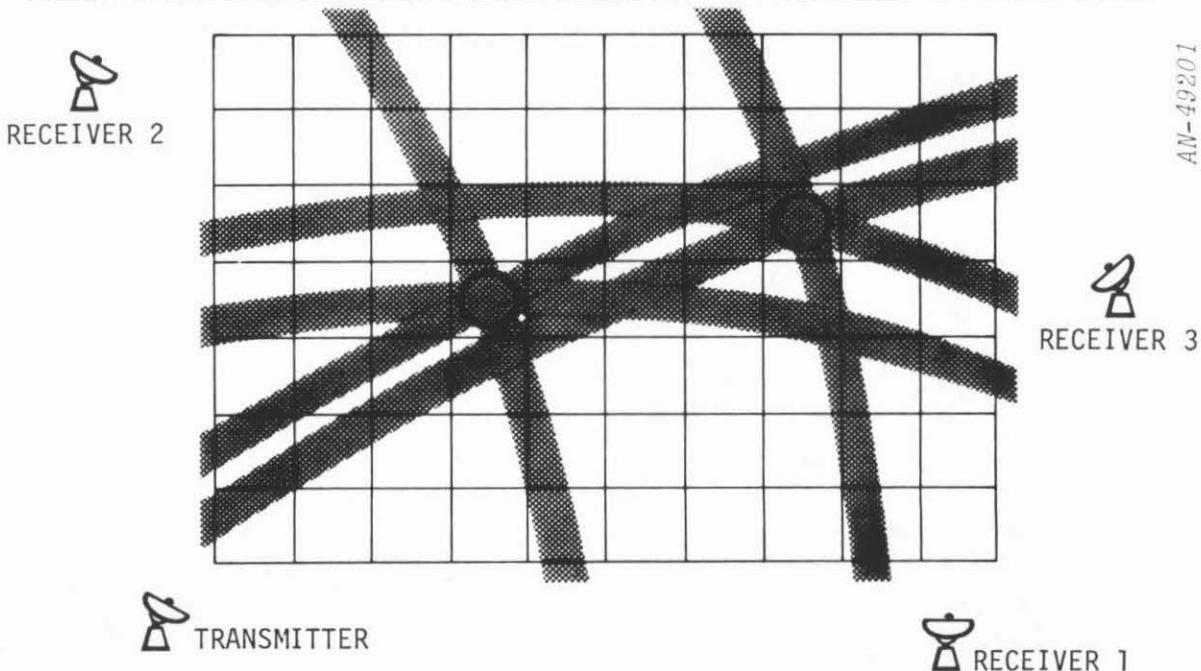


Figure 3. In Solution One, space is divided into regular cubes, and a LEM element is associated with each. The LEM element computes the number of ellipsoids that intersect within its cube.

around this concept was designed and roughly simulated. It was rejected because it was found that adequate performance required approximately 10^{10} elements (10^{13} bits), a number well beyond practical near-term development. In addition, this approach resulted in less than 1% of the elements reporting hits, even though the noise of the radars resulted in many elements whose cubes were outside the transmitter's beam reporting hits. It seemed clear that the information whether or not the cube was in the transmitter's beam should be added to the hit count.

2.3 SOLUTION TWO

An obvious way to do this was to subdivide only the volume of the transmitter's beam into cells--each cell being identified with a LEM element. Whenever the beam position changed, the appropriate LEM cells would be reloaded with precalculated parameters expressing the geometric relationship of the new transmit beam to whichever set of six radar receivers was selected.

To be more specific, the transmitter's beam was divided radially into 2048 "rays" emanating from the transmitter as in Fig. 4. Associated with each ray was a LEM element consisting of six calculation circuits (one for each receiver), six comparison circuits, and a 5-out-of-6 comparator to examine the six comparison circuits for 5 out of 6 "hits" (see Fig. 5). Individual hits were discovered through the following mechanism: we observed that if the different bistatic ranges for an object (obtained by the different receivers) corresponded to a real target (an intersection of 5 out of 6 ellipsoids), then each of the different bistatic ranges would correspond approximately to the same distance, r , from the transmitter. The relationship between r_j , the distance from the transmitter to the target along the j th ray, and B_i , the bistatic range measured by the i th receiver, can be shown to be:

$$r_j = \frac{1}{2} \frac{B_i^2 - P_{ij}^2}{B_i - C_{ij}} \quad (1)$$

where P_{ij} and C_{ij} are fixed geometric parameters relating the j th ray in the transmitter's beam to the i th receiver.

Given this, it was only necessary for each of the six calculation circuits in each ray to solve Eq. 1 for r_j --given each bistatic range, B_i , it received, and the comparison circuit to check whether five out of six values for r were "close enough." If so, a hit was recorded and a target located. In this design, then, a LEM element is identified with a volume of space corresponding to an elemental ray. Then each LEM element is divided into six smaller subelements, one for each of the six receivers.

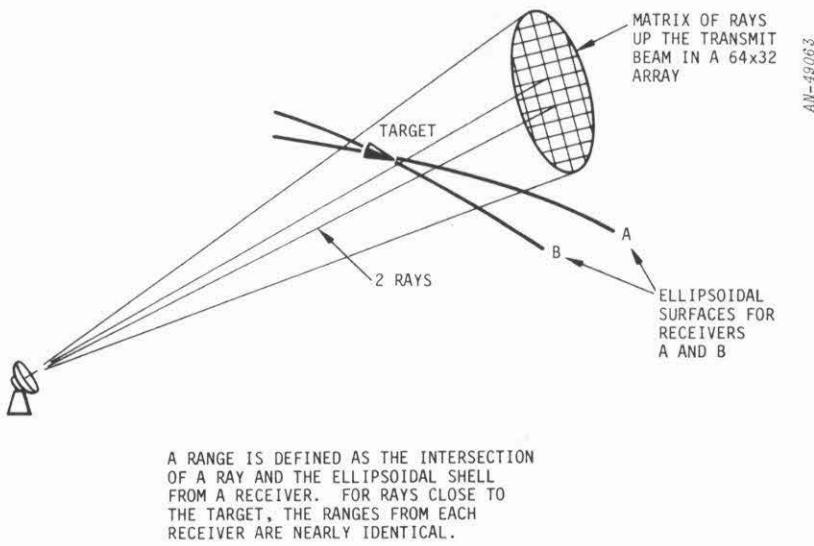


Figure 4. In Solution Two, a range is defined by the intersection of a "ray" and the ellipsoidal shell from a receiver. For rays close to the target, the ranges from each receiver are nearly identical

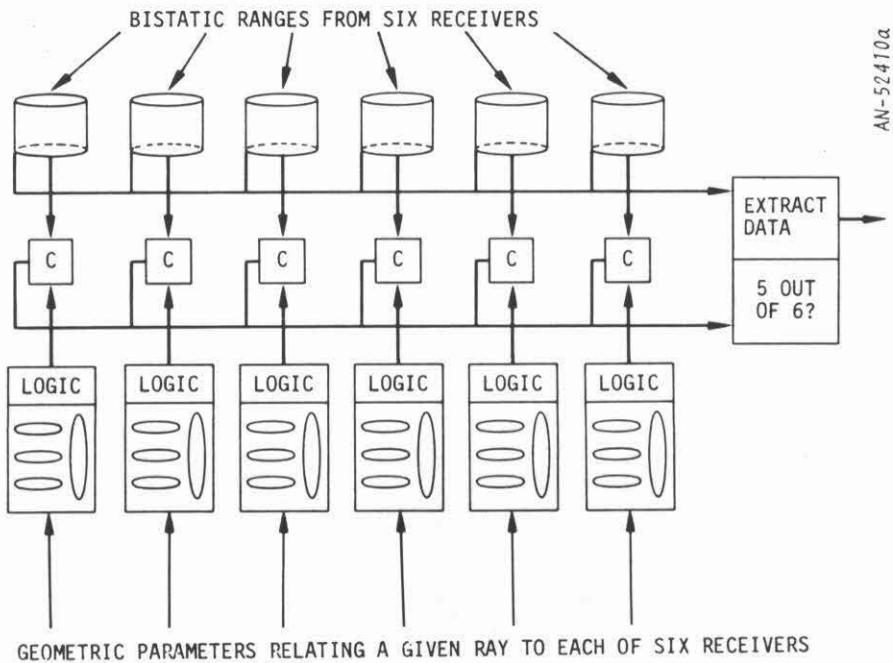


Figure 5. A Single LEM Element for Solutions Two and Three

Simulation of this design indicated that in a realistic scenario with about 40 targets in the beam, the LEM would reject about 99% of the 64,000 ghosts in the beam, and pass 560 ghosts along with the 40 real targets to a postprocessor that would eliminate the remaining ghosts. These residual ghosts are produced by the effects of noise in the radar signals, and are removed by fairly standard signal extraction techniques on a sequential processor (a PDP-11/45). Since relatively small numbers of these residual ghosts were passed to the sequential processor, we did not examine the possibility of removing them with a LEM-like device.

2.4 SOLUTION THREE

Solution Two was on the verge of prototype manufacture when a simpler design was discovered. First, notice that Eq. 1 is monotonic in the bistatic ranges B_i , which arrive at the receivers in ascending order because returns from the closest targets arrive before those from the more distant ones. This implies that if we view the problem in terms of the ranges r from the transmitter to the target, then the bistatic returns arrive in the order of their range distance along the appropriate ray.

This implies that if each ray were to be scanned in an ascending manner beginning at the transmitter, the set of bistatic ranges does not have to be randomly accessed to discover which B_i 's correspond to a given range r . Instead, it is only necessary to store the set of B_i 's in a FIFO memory, with one FIFO memory for each receiver for each ray. This observation not only allows us to use a simple memory structure for the bistatic range buffers of Fig. 5, but also suggests a way to simplify the logic of each of the six calculation circuits in each ray-associated element. A finite difference method can be used to "scan along" each ray.

Each of the memories of the six logic elements in each LEM element is loaded with a set of parameters describing the first, second, and third differences of the expression in Eq. 1. Then, by simple addition, the corresponding ΔB_i can be calculated for each Δr along the ray. Thus, the logic in each of the six elements can be replaced with the simple "difference engine" logic of Fig. 6.

In operation the initial bistatic range is loaded into register A, its rate of change with respect to r into B, the rate of rate of change into register C, and the rate of rate of rate of change into D. For each incremental advance along the ray, the clock line is asserted, causing the following operations:

$$\begin{aligned}C &\leftarrow C+D \\B &\leftarrow B+C \\A &\leftarrow A+B\end{aligned}$$

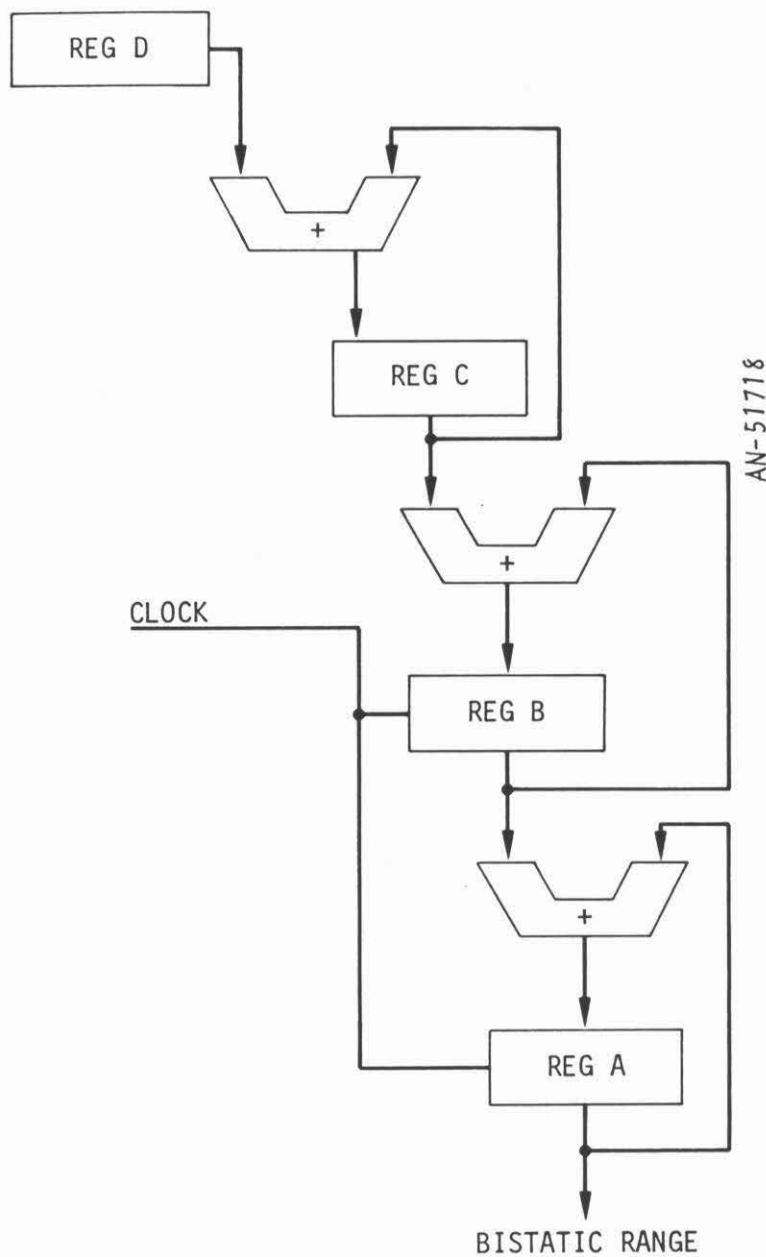


Figure 6. A Target Finder Difference Engine

(Simulation of this design under realistic scenarios has indicated appropriate register sizes and their initial values. More details are given in a General Research Corporation contract report.¹

Now, at each step along the range r , the corresponding window of bistatic ranges ΔB_i can be calculated by each of the six logic elements in a ray-associated LEM element. The received bistatic ranges in the FIFO memories are compared with the calculated ΔB_i by each of the six comparison circuits and a "probable target" is reported if five of the six comparators report a hit. The 5-out-of-6 condition only indicates a probable target because the noise in the received bistatic returns causes some bistatic returns to only apparently fall within the required window. This effect produces some false targets which are then eliminated by a PDP-11/45 class postprocessor. High-fidelity simulations of this LEM target finder design indicate that with 40 targets in the field of view, this LEM target finder reduces the 64,000 ghosts to a more manageable 600 false targets.¹ Thus, this simple collection of 12,288 Babbage-like differential engines distributed over about 10^{12} bits of memory reduces the calculation load from one requiring about twenty PEPE processors to one requiring only a small minicomputer.

A single element for such a "difference engine" LEM target finder is being constructed now² and will be installed at the BMD Advanced Technology Center at Huntsville, Alabama, during the second quarter of 1979 for more exhaustive testing.

2.5 CONCLUSIONS

The following table illustrates some of the major design properties of the three logic-enhanced memory designs. (In fact, two additional LEM designs were developed for the target finder problem, but in the interest of brevity they are not reported here.)

Solution One utilizes considerable memory and very little logic to process the data in each element. Its relatively slow performance is due primarily to the time required to load such an enormous number of elements with bistatic range information and interrogate them for "hits". Such an architecture is communication-bound.

Solution Two uses the least memory and the most complex logic per memory element. The solution exhibits the most processor-intensive architecture and is limited by the processing time of the bit-slice microprocessor used to implement the design.

¹E. Hatt and H. Ostrowsky, Logic-Enhanced Memory: Final Report, Vol. I, "The LEM Target Finder," General Research Corporation CR-2-776 (Contract DASG60-77-C-0066), July, 1978.

²By Honeywell Advanced Systems, Minneapolis, Minnesota.

	Number of LEM Elements	Memory Re- quired, Bits	Logic Required Per Element	Time to Deghost 40 Returns, ms
Solution One	10^{10}	10^{13}	1 compare	40
Solution Two	12,288	10^6	1 multiply 1 divide 2 adds 1 compare	3
Solution Three	12,288	10^9	2 adds 2 compares	0.2

Solution Three uses only a little more logic per element than the communication-bound Solution One. Its speed is memory-bound, limited by the transfer rate from the serial-access bubble memories.

These three architectures each implement a different algorithmic approach to a single problem, and each one has practical performance limits imposed by a different aspect of the architecture. All three designs, however, utilize large amounts of memory in distinctly non-von Neumann ways to reduce the processing load on a conventional processor.¹

Future research will be directed toward characterizing these unusual memory architectures and the algorithms that utilize them. We are also exploring other computationally stressing problems in ballistic missile defense to discover logic-enhanced memory architectures which can substantially relieve these problems.

¹For a more detailed description of several LEM designs applied to the radar deghosting problem, see E. Hatt and H. Ostrowsky, op. cit.

WSI Distributed Logic Memories

R.M. Lea and M Sreetharan
Brunel University.

INTRODUCTION

Advances in semiconductor technology, which have led to VLSI, are providing dramatic improvements in speed, cost and reliability of computer hardware components. However, in the future such improvements (viz. faster and larger stores, faster processors etc) will be limited by

1. the basic SISD (Serial Instruction - Serial Data) computer architecture, which has remained relatively unchanged from its original conception by Babbage in 1832 and its engineering specification by von Neumann in 1946, and
2. its means of implementation, which incurs the expensive overheads of printed circuit board layout, assembly and testing.

In view of these restrictions it is expedient to channel semiconductor development towards experimentation with new computer architectures.

Wafer-Scale Integration¹ (WSI) offers the possibility of departing from the von Neumann computer architecture and alleviating its implementation problems. By interconnecting the good chips on an undiced wafer, WSI provides a multiplicity of processing elements and bypasses the expensive stages of chip and printed circuit board manufacture. Whereas VLSI offers low-cost components at the sub-system level, WSI offers low-cost computer systems. Hence, traditional market pressures deter the speculative development of a range of VLSI chips in order to launch a radically new computer structure. However, WSI offers the integration of a new architecture in a single development.

WSI has attracted computer system designers since its inception by Petritz² as 'discretionary wiring' in 1967, and various such schemes have been considered since that time. Several techniques for selecting good chips emerge from this work. Discretionary wiring uses a second level of metallisation to interconnect those chips passing wafer probe tests. A model of the ALAP (Associative Linear Array Processor) proposed by Finnilla and Love³, and based on discretionary wiring, has been built and tested by Hughes Aircraft. Another technique proposed

by Elmer, Tchon, Denboer, Frommer, Kohyama, Hirabayashi and Ngina⁴, uses fusible links to avoid the expense of preparing a special metallisation mask for each wafer. Other techniques proposed by Catt⁵ and Manning⁶, use a standard metallisation mask, but each chip comprises fault tolerance logic which enables a chain of good chips to be created. A hardware model of the former has been investigated by Aubusson^{7,8} at Middlesex Polytechnic. The common architectural feature of these WSI designs is the construction of a very long segmented shift register.

Initial considerations indicate that WSI could provide the means of ⁹ implementing the Distributed Logic Memory originally proposed by Lee in 1962, modified by Paull¹⁰ and Gains¹¹ and extended by Savitt, Love and Troop¹², Kisylia¹³, Sturman¹⁴, Lipovski¹⁵, Wright^{16,17}, Beavan¹⁸, Leal¹⁷ and Lewin^{16,18,19}. In view of this possibility, it is interesting to speculate on the influence WSI may have on Computer Architecture.

At Brunel University, the systems, software and application aspects of WSI are being investigated in an ACTP funded contract. A class of distributed logic memories suitable for fabrication with WSI techniques has been identified and a software research vehicle is being constructed for experimental investigations. Two specific distributed logic memories based on a proposal by Catt²⁰ have been specified for feasibility studies in text compression applications. This paper outlines the design philosophy, structural organisation and operational principles of a general member of the class of distributed logic memories for WSI fabrication.

A DISTRIBUTED LOGIC MEMORY FOR WSI FABRICATION.

The architecture of a machine suitable for fabrication with WSI is described below. The structure of this machine is not designed for a specific application, but is intended to illustrate the architectural potential of WSI distributed logic memories.

The machine is a distributed logic memory, organised as a long string of identical cells (viz. processing elements), interconnected by two communication lines, the 'fast line' and the 'slow line', which at the ends of the string are connected to a host processor as shown in Fig.1.

The host processor acts as the source and sink of information (viz. instructions and data) for the communication lines. Information flows unidirectionally along the string and is processed bit-serially in transit without recourse to an external processing unit. Thus,

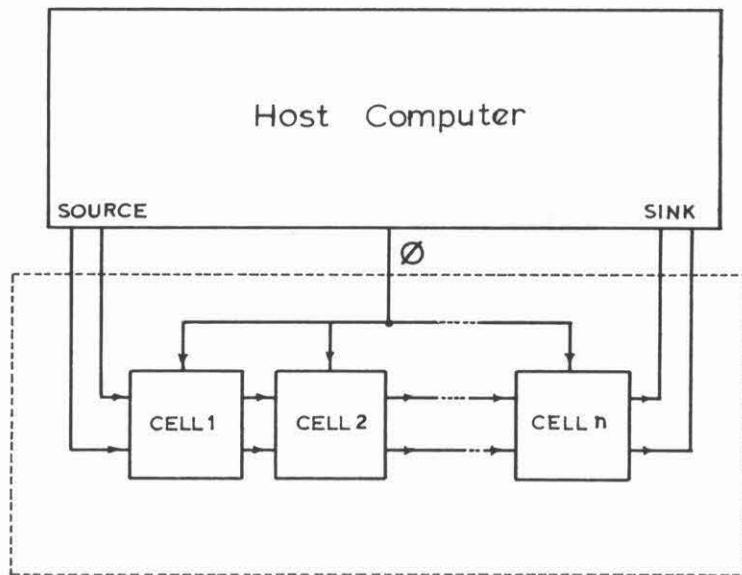


Fig 1. WSI Distributed Logic Memory

each cell acts as a sink for information flowing from upstream cells and a source for downstream cells relative to its position in the string. Thus, the slow and fast line inputs (S_i and F_i) of each cell are coincident with the slow and the fast line outputs (S_o and F_o) of its upstream neighbour.

Information on the lines is represented by fixed-length (viz. N-bits) instruction and data words. Each cell comprises storage for one data word in the slow line whereas the fast line supports a relatively unimpeded flow of instruction words. Thus, for each data word supported by the slow line, the fast line appears to support a sequence of instructions as indicated in Figures 2 and 3.

A key feature of the machine architecture is its content-addressing capability and the instruction and data words are formatted accordingly. Each N-bit word comprises three major fields, namely the tag field, the command field and the data field. Optional fields and special bits may be used to increase the operational capability of the machine.

Instruction word :

data field	command field	tag field
N - 1		0
data field		tag field

Instruction execution is preceded by comparison of the tag fields of the instruction and data words. If there is a match in cell i , the operation specified in the command field of the instruction word is executed in that cell on the operands in the data fields of the instruction and data words and the result is sourced to the downstream cell $i + 1$. If the tags mismatch, then the instruction is not executed in cell i . Thus, instruction execution is conditional on the content of the tag field of data words as illustrated in Fig. 2.

Each cell comprises two major functional units.

1. the Information Flow Network (IFN)
2. the Local Control Unit (LCU)

as shown in Fig. 4.

Bit time

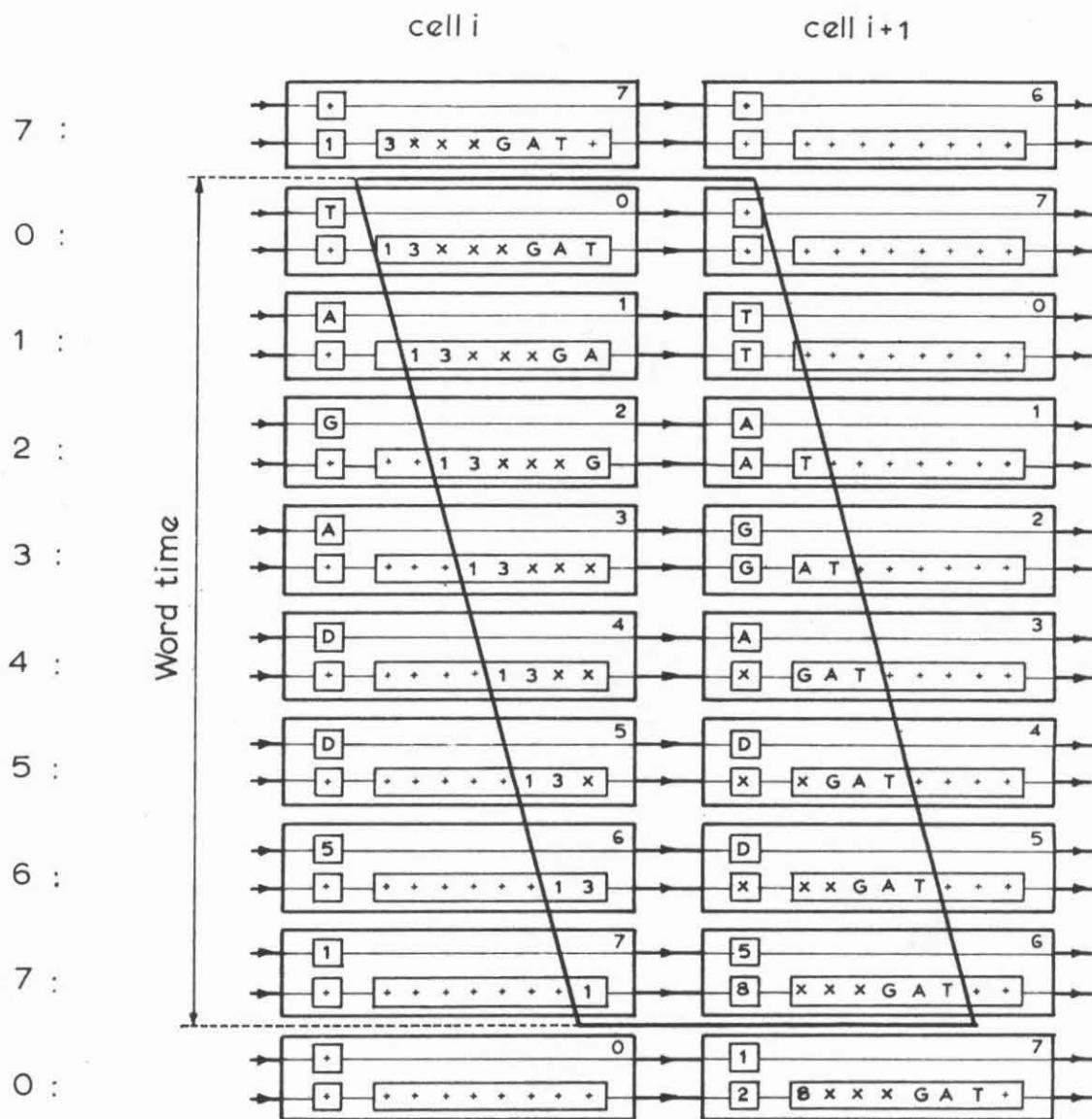


Fig 2. Data flow on the slow line

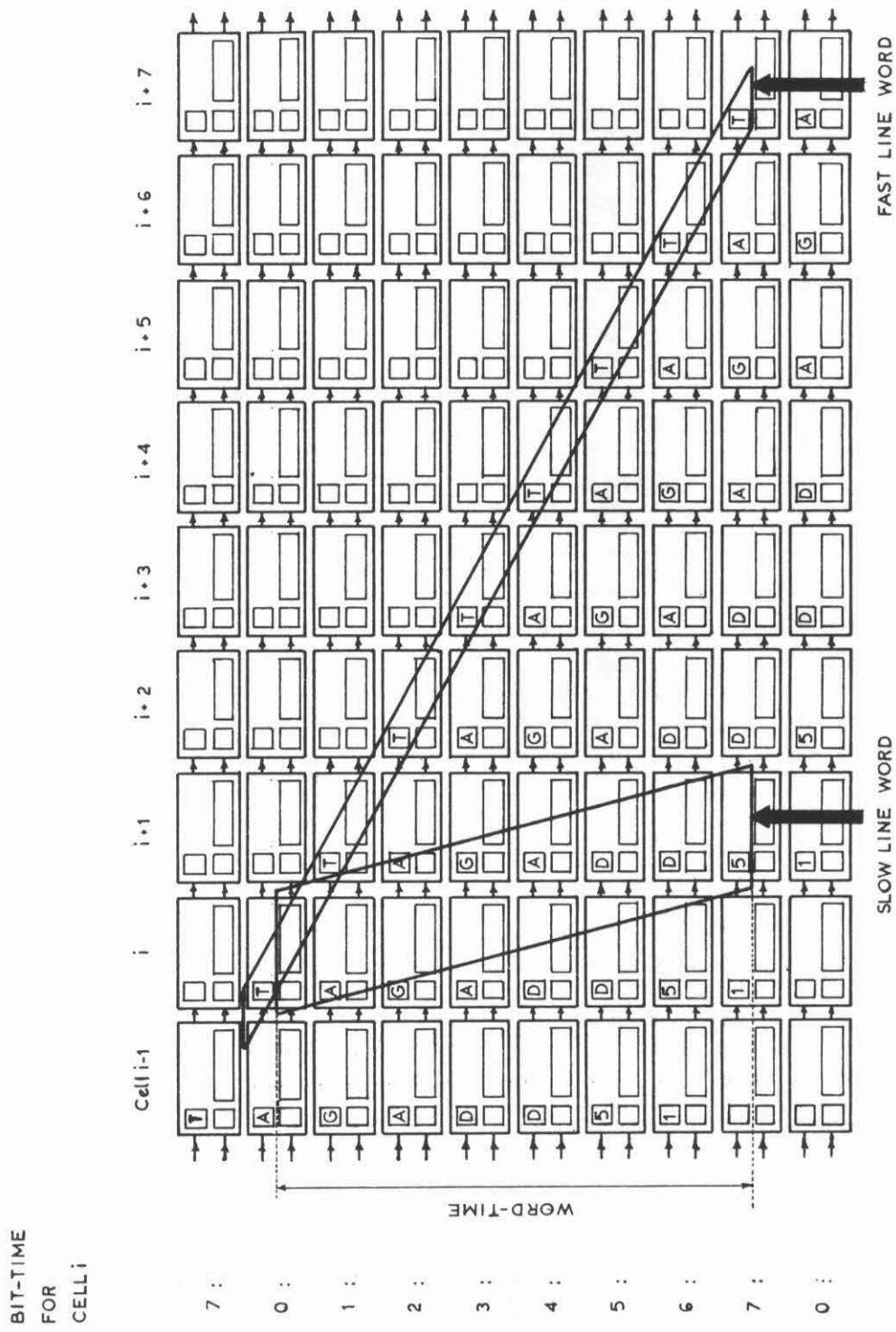
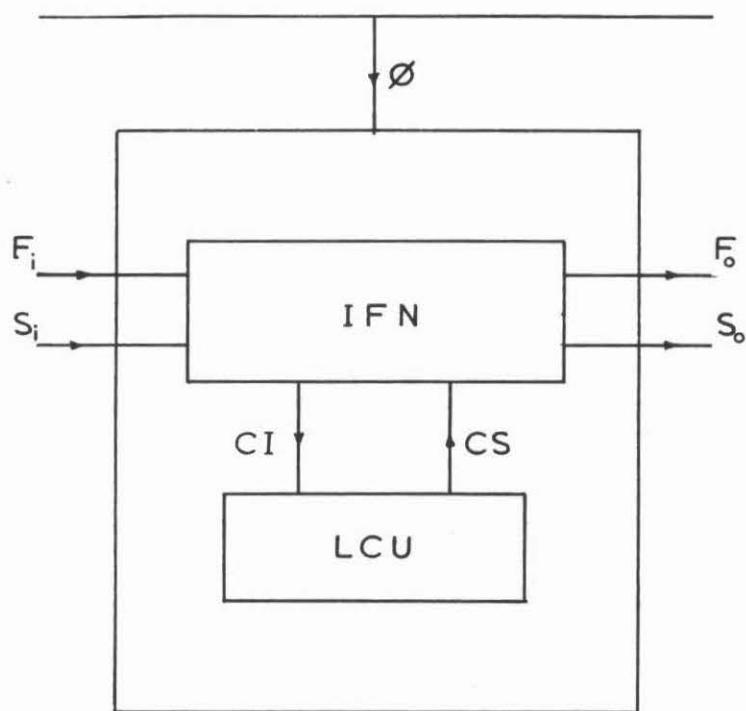


Fig 3. Data flow on the fast line relative to the slow line

**Legend**

IFN	-	Information Flow Network
LCU	-	Local Control Unit
F_i	-	Fast line input
S_i	-	Slow line input
F_o	-	Fast line output
S_o	-	Slow line output
CI	-	Control Information
CS	-	Control Signals
\emptyset	-	Global clock

Fig. 4. Major functional units of a cell

The Information Flow Network (IFN) recognises instructions and where the contents of the tag field of an instruction word matches the contents of the tag field of the stored word, it routes the cell command to the Local Control Unit (LCU) which executes cell operations according to the state of a resettable counter which governs the timing within the cell.

The IFN includes a 1-bit buffer for each of the fast lines and slow line inputs (F_i and S_i) and an N-bit serial-in, serial-out shift register which provides local storage for an N-bit word. The IFN performs,

1. Flexible routing of the different fields of the instruction and data words between the fast line, slow line and the shift register.
2. arithmetic and logic operations on the data-fields of the stored data word and the incoming instruction word.

The data word stored in the shift register of a particular cell can be moved relative to the data words in the shift registers of other cells as follows:

1. Rotation of the data word within the shift register has the relative effect of progressing the word 'upstream'.
2. Transferring the data word from the shift register to the fast line has the relative effect of progressing the word 'down-stream'.

CONCLUSIONS

Wafer-Scale Integration¹⁻⁸ (WSI) could provide a means of implementing cost-effective distributed logic memories which have been of interest for nearly 17 years. Although the hardware feasibility of WSI is not yet proven, sufficient work has been done to indicate that innovative wafer assembly and packaging techniques would lead to cost-effective WSI devices. Thus, it is interesting to speculate on the influence WSI distributed logic memories might have on computer architecture.

In contrast to the von Neumann computer architecture, distributed logic memories are data-flow machines. The cpu of the former is specifically programmed to execute a sequence of operations on essentially unordered and implicitly addressed operands. Thus, the conventional machine is based on an ordered instruction-flow to an explicitly referenced single-processor, such that only one operation can be performed on only one or two operands at any given time. However, the distributed logic memory

supports a programmed data flow within which sequentially ordered operands are implicitly addressed and manipulated without reference to specific cells. Thus, the distributed logic memory is based on an ordered data-flow within an implicitly-referenced multi-processor string such that many different operations can be performed on many different operands at the same time. Consequently, WSI distributed logic memories are only suitable for those applications where large data blocks can be processed in transit.

There are three application classes where a WSI distributed logic memory could be inserted in a data stream for 'on-the-fly' information processing.

1. As a front-end processor, WSI distributed logic memories could be incorporated in the communication channels of computer networks for special purpose tasks such as error detection, data compaction, code translation etc.
2. As a main-frame processor, the intrinsic parallelism of WSI distributed logic memories offers considerable benefits for string processing algorithms. Hence, WSI distributed logic memories should be well suited to sampled data processing (viz. signal correlation, digital filtering, fast fourier transform etc) and text string processing (viz. text editing, lexical analysis etc.) However, it is unlikely that WSI distributed logic memories will benefit conventional main-frame processing techniques involving random store accesses.
3. As a back-end processor WSI distributed logic memories could be incorporated in peripheral equipment controllers for file searching updating and maintenance in support of data-base management and information retrieval systems.

Although the application environment of WSI distributed logic memories is well developed the structural organisation and programming philosophy of such devices are still matters of research. Accordingly, software simulations, written in Pascal, of specific WSI distributed logic memory structures are under investigation at Brunel University. In addition, the feasibility of WSI distributed logic memories for on-line text compression, for efficient data communication and storage systems, is being investigated, as part of an ACTP funded contract.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the support of ACTP (Department of Industry) and the many fruitful discussions with R.C. Aubusson, I. Catt and E.A. Newman.

REFERENCES

1. R.C. Aubusson and I. Catt, "Wafer-Scale Integration A New Approach". ESSCIRC Digest 77, pp. 76-78, Sept. 1977.
2. R.L. Petritz, "Current status of LSI technology", IEEE. J. of Solid-State Circuits, vol. SC-2, pp. 130-147, Dec. 1967.
3. C.A. Finnila and H.H. Love, "The Associative Linear Array Processor", IEE Trans. on Computers, vol. C-26, No. 2. pp. 112-125, Feb, 1977.
4. B.R. Elmer et al., "Fault tolerant 92160 bit multiphase CCD memory", IEEE Int. Solid-State Cir. Conf. Dig., pp. 116-117, Feb, 1977.
5. I. Catt, "Improvements relating to digital integrated circuits", British Patent Specifications 1 377 859, Dec, 1974.
6. F.B. Manning, "An approach to highly integrated, computer-maintained cellular arrays", IEE Trans. on Computers, vol. C-26, pp. 536-552.
7. R.C. Aubusson and I. Catt, "Wafer-Scale Integration - A Fault-Tolerant Procedure", IEEE J. of Solid-State Circuits, vol. SC-13, pp. 339-344, 1978.
8. R.C. Aubusson and R.J. Gledhill, "Wafer-Scale Integration - Some Approaches to the Interconnection Problem", Microelectronics, vol.9, No.1, pp. 5-10, 1978.
9. C.Y. Lee, "Intercommunication cells - bases for a distributed logic computer", Proc. AFIPS (FJCC), 22, pp. 130-136, Dec, 1962.
10. C.Y. Lee and M.C. Paull, "A content addressable distributed logic memory with applications to information retrieval", Proc. IEEE, 51, pp. 924-932, June 1963.
11. R.S. Gains and C.Y. Lee, "An improved cell memory", IEEE Trans. on E.C., 14, pp. 72-75, 1965.
12. B.A. Savitt, H.H. Love and R.E. Troop, "ASP - a new concept in language and machine organisation", Proc. AFIPS (SJCC), 30, pp. 87-102, 1967.

13. A.P. Kisylia, "An associative processor for information retrieval", Co-ordinated Science Lab. (Illinois University), Report R-390 (AD 67531O), Aug, 1968.
14. J.N. Sturman, "An iteratively structured general purpose digital computer", IEE Trans. EC-17, pp. 2-9, 1968.
15. G.P. Lipovski, "The architecture of a large associative processor", Proc. AFIPS (SJCC), 36, pp. 385-396, 1970.
16. J.S. Wright and D.W. Lewin, "A draft specification for a symbol processor", IEE Conf. Computer Science and Technology, (IEE Pub. No. 55) pp. 282-295, 1969.
17. R.M. Lea and J.S. Wright, "A Novel memory concept for information processing", Datafair Research Papers, 11, pp. 413-417, 1973.
18. P.A. Beavan and D.W. Lewin, "An associative parallel processing system for non-numerical computation", The Computer Journal, 15, 4, pp. 343-349, 1973.
19. D.W. Lewin, "Introduction to Associative Processors", Proc. NATO Advanced Study Institute on Computer Architecture 12-14 Sept, 1976, St. Raphael, France, pp. 217-234.
20. I. Catt, "Specification of Property la invention", Aug, 1978.

SYSTOLIC PRIORITY QUEUES

Charles E. Leiserson

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

Copyright -C- 1979 by Charles E. Leiserson

Reproduced by Permission

This research is supported in part by the National Science Foundation under Grant MCS 75-222-55, the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422, and by the Fannie and John Hertz Foundation.

1. Introduction

Very large scale integrated (VLSI) circuit technology has made it possible to build multiprocessor hardware devices to aid in the rapid solution of sophisticated problems. An algorithms designer wishing to take full advantage of the massive parallelism offered by VLSI must address geometric issues hitherto relegated to layout artists. The reason for this is that VLSI is a planar technology in which the interconnections among components on a chip may cost more than the components themselves. The designer of a multiprocessor algorithm to be implemented in this technology must consider the complexity of the data paths between processors in evaluating the algorithm.

Many programming applications require the ability to insert *records* into a set, and at any time to retrieve from the set the record having the smallest *key* according to some ordering. A data structure that provides such services is called a priority queue. (See Knuth [1973], pp. 150-152 and Aho, Hopcroft, and Ullman [1974], pp. 147-152.) The operation $\text{INSERT}(Q, a)$ replaces the set Q with the set $Q \cup \{a\}$. The operation $\text{EXTRACT_MIN}(Q)$ returns the smallest element a of Q and replaces Q with $Q - \{a\}$. This paper shows how high-performance priority queues can be built using the VLSI technology.

Section 2 of this paper discusses systolic systems, the model of parallel computation used for this work. Section 3 presents a systolic array implementation of a priority queue. Section 4 shows how multiple priority queues can be implemented as a single device that shares processors among the queues. The organization of the shared structure is presented in Section 5. Section 6 deals with the geometric layout of the multiple queue device in VLSI. The conclusion is presented in Section 7.

2. Systolic Systems

A systolic system is a network of processors that rhythmically compute and pass data among themselves. The analogy is to the rhythmic contraction of the heart which pulses blood through the circulatory system of the body. Each processor in a systolic network can be thought of as a heart that pumps multiple streams of data through itself. The regular beating of these parallel processors keeps up a constant flow of data throughout the entire network. As a processor pumps data items through, it performs some constant-time computation and may update some of the items.

Systolic systems provide a realistic model of computation which captures the concepts of pipelining, parallelism, and interconnection structures. Kung and Leiserson [1978] demonstrates that many basic matrix computations can be performed by systolic systems whose underlying network is array structured. These systolic arrays are suitable for implementation as VLSI hardware devices. This paper will show the utility of systolic trees.

Unlike the closed-loop circulatory system of the body, a systolic computing system usually has ports into which inputs flow, and ports from which the results are retrieved. Thus a systolic system can be a pipelined system - input and output occur with every pulsation. This makes them attractive as peripheral processors attached to the data channel of a host computer. Figure 1 illustrates how a special-purpose systolic device might form a part of a PDP-11 system. A systolic system might be attached directly to the CPU of a Von Neumann machine, much as a floating-point processor may be added to extend the instruction set of a computer.

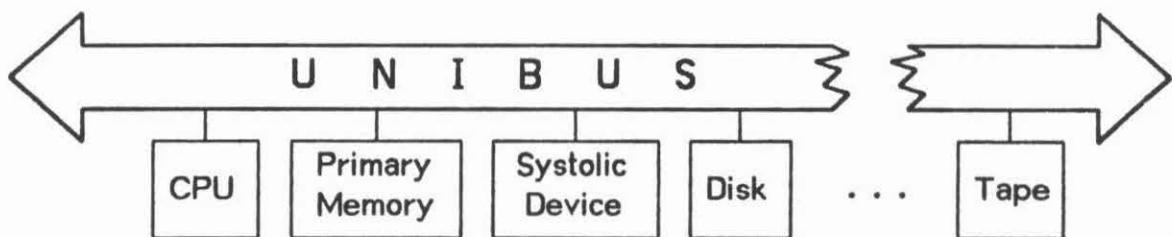


Figure 1: A systolic device connected to the UNIBUS of a PDP-11.

The activities of the processors in a systolic system can be assumed to be synchronous. With each pulse of a clock, a processor executes the same constant-time program. Furthermore, each processor is only allowed a fixed number of input and output lines and a constant amount of local storage. It is possible to view the processors as being asynchronous, each computing its output values when all its inputs are available, as in the data flow model. For the results of this paper, the synchronous approach is more direct and intuitive.

3. A Simple Systolic Priority Queue*

A linear systolic array can implement a fast priority queue. Each processor in this array has two registers A and B, and each processor can access the registers of its two neighbors, as shown in Figure 2. The A registers hold elements in the queue in sorted order, with the smallest element in A_1 . The B registers contain elements that are being inserted into the queue. Initially, all the elements in the queue are $+\infty$. The priority queue operations INSERT and EXTRACT_MIN are performed by the user at the left end in the diagram. As items are inserted by the host, they displace overflow elements which are output at the right end. Normally, the overflow element will be $+\infty$, but when this is not the case, a real overflow has occurred.

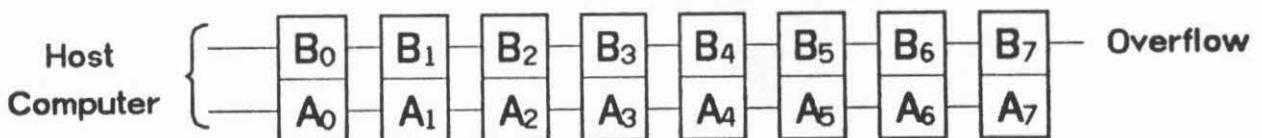


Figure 2: A simple systolic priority queue.

Even and odd numbered processors alternately pulsate, each time executing the following:

1. $B_i \leftarrow B_{i-1}$.
2. Arrange the elements in A_{i-1} , A_i , and B_i so that $A_{i-1} \leq A_i \leq B_i$.

Processor 0 is a dummy processor which does not execute any code, but whose registers can be altered by the host machine. The array pulses twice each time an operation is performed by the host machine, once for odd numbered processors and once for those with even positions. The operation $\text{INSERT}(Q, a)$ is implemented by placing the item a in B_0 and $-\infty$ in A_0 just before processor 1 pulses. Each element travels to the right until it finds its place in the array.

By loading A_0 and B_0 with $+\infty$, the pulsation of the systolic array causes

*This section describes research done jointly with H.T. Kung.

	0	1	2	3	4	5	6	7	Step
B	6		9		15		∞		0 INSERT(6)
	$-\infty$		3	5	10	14	∞	∞	
A		6		9		15		∞	1.1
	$-\infty$	3	5	10	14	∞	∞	∞	
		6		10		∞		∞	1.2
	$-\infty$	3	5	9	14	15	∞	∞	
			6		10		∞		2.1
		3	5	9	14	15	∞	∞	
	∞		6		14		∞		2.2 EXTRACT MIN
	∞	3	5	9	10	15		∞	
		∞		6		14		∞	3.1
	∞	3	5	9	10	15	∞	∞	
		∞		9		15		∞	3.2
	3	∞	5	6	10	14	∞	∞	
		∞		9		∞		∞	4.1
		∞	5	6	10	14	∞	∞	
	8		∞		9		15		4.2 INSERT(8)
	$-\infty$	5	∞	6	9	14	15	∞	

Figure 3: Several steps in the execution of the systolic array shown in Figure 2.

EXTRACT_MIN to be performed, the minimum value being found in A_0 . With each pair of pulsations the systolic array is ready to execute another INSERT or EXTRACT_MIN operation. Figure 3 shows several steps in the execution of this systolic array. The initial configuration in the figure shows the insertion of items 9 and 15 already in progress. Although it may take an element a long time to find its place in the systolic array, to the host computer an INSERT operation appears to take only constant time. Since the minimum element in the queue is always at the front, an EXTRACT_MIN operation also appears to take constant time. The operation of the systolic array is pipelined so that no degradation occurs even when the host executes many priority requests in a row. Thus we may say that the systolic array has a response time which is a constant, independent of the length of the array.

4. The Systolic Multiqueue

Suppose several of the simple priority queues in Section 3 are attached as a device to a host computer. No matter how a fixed number of processors are allocated, the capacity of any particular queue may be exceeded while most of the other queues are empty. In this section a single device is presented that is capable of implementing many priority queues that dynamically share processors. Like the simple queue in the previous section, the systolic multiqueue can perform INSERT and EXTRACT_MIN for a single host computer, on any of m queues, with a response time that is a constant, independent of the size of the queue.

Figure 4 illustrates the organization of the systolic multiqueue. Each of the m queues to be implemented requires a systolic array of the type presented in Section 3. These can be accessed directly by the host computer. When a systolic array overflows, the overflow element travels through a switching network to a large systolic tree. Each time the minimum of a particular queue is extracted from the corresponding systolic array, the minimum of the elements that have overflowed from that queue is removed from the systolic tree. The internal structure of this shared overflow area is examined in Section 5. Here, we only need to know its behavior.

The records stored in the systolic tree are the same as those in the systolic arrays, but an additional field is used to identify the queue from which the item originated. Thus items are stored in the systolic tree according to a composite record $\langle Q, a \rangle$ where a was originally inserted by an $\text{INSERT}(Q, a)$ operation and eventually overflowed from the systolic array corresponding to that queue.

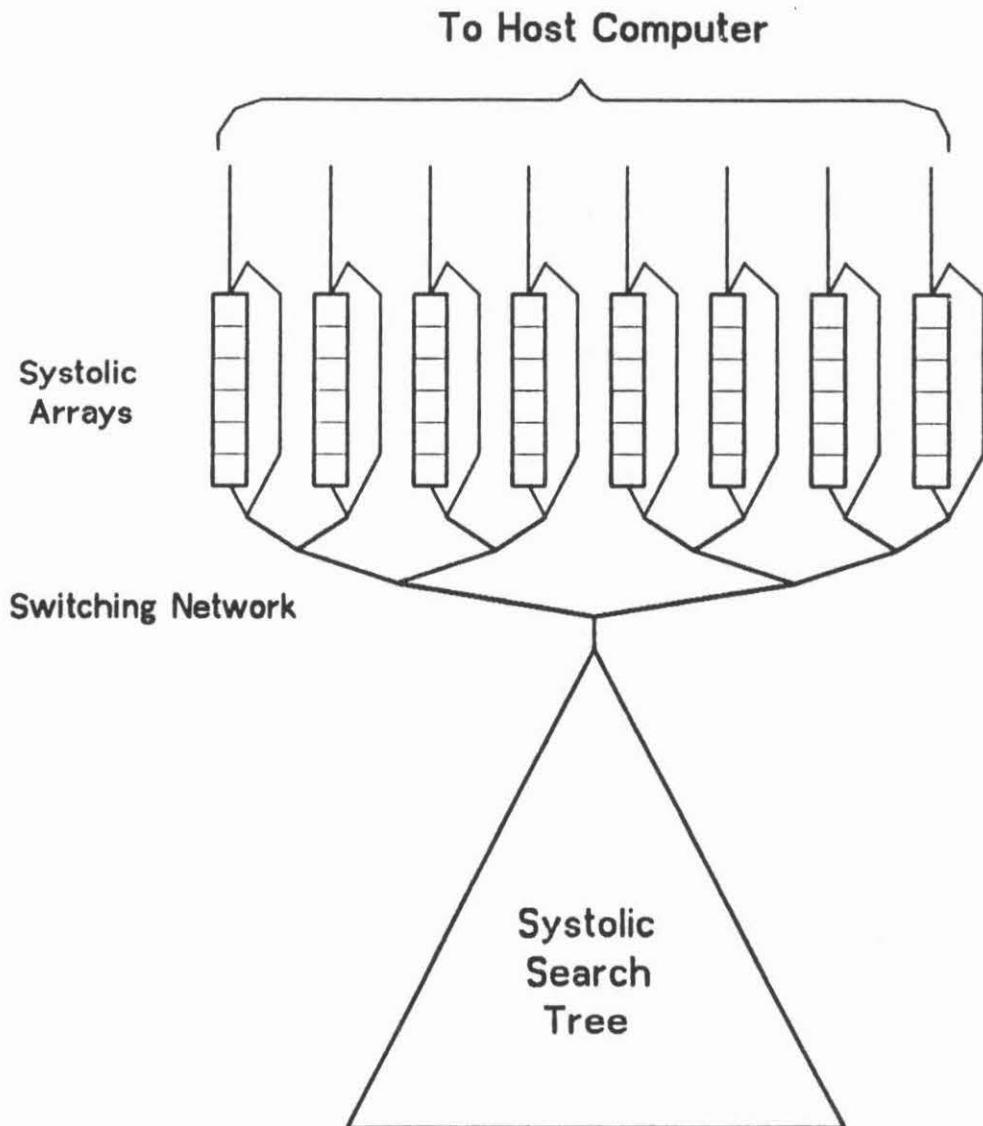


Figure 4: The systolic multiqueue device.

The operations that can be performed by the systolic tree are very similar to those commands given to the entire systolic multiqueue by the host. The composite record $\langle Q, \alpha \rangle$ is inserted into the systolic tree by `INSERT(Q, \alpha)`, and `EXTRACT_MIN(Q)` removes the

smallest element in the systolic tree which has Q as its first field. As will be seen in Section 5, a systolic tree of size n can perform each operation in time $O(\log n)$. The operations are pipelined, however, so that the systolic tree can process several operations in parallel, waiting only constant time between successive operations. For example, if a sequence of EXTRACT_MIN's are started constant time apart, it will take $O(\log n)$ for the first minimum to be retrieved, but then results appear with every cycle of the systolic tree. Thus the systolic tree provides high throughput, with $O(\log n)$ response.

The claim was made, however, that the systolic multiqueue had constant response time for each of m queues. Systolic arrays of size proportional to $\log n + \log m$ are used to achieve this goal by satisfying any immediate requests from the host. When the host executes EXTRACT_MIN(Q), that operation is performed on the corresponding systolic array. At the same time, a request is put into the systolic tree to perform an EXTRACT_MIN(Q). A result is yielded by the systolic tree $O(\log n)$ time later and takes $O(\log m)$ more time to traverse the switching network. It is then inserted into the systolic array at the same end the host computer uses. Even if the host has performed $\log n + \log m$ EXTRACT_MIN(Q) operations in the meantime, the quick response systolic array has been able to satisfy the requests. Now if the host continues to perform EXTRACT_MIN(Q)'s, a stream of results from the systolic tree will be inserted into the array just in time to satisfy the requests. The systolic array will always have at least one item in it because operations on the systolic tree are pipelined. It does not matter whether or not the host accesses different queues. Since it can only access one queue at a time, no systolic array will empty before the beginning of a stream of items from the systolic tree has reached the systolic array.

The number of processors in the systolic arrays is $m \log n$. If the size of the systolic tree is doubled, this means only m more processors need be added to the systolic arrays. The amount of sharing of processors among the m queues is clearly substantial. Furthermore, the systolic multiqueue will not overflow until the shared systolic tree overflows. In fact, overflow of the systolic tree can be handled "nicely" as will be seen in Section 5.

5. Systolic Trees

It seems natural to use a tree-structured hardware device to achieve pipelined performance with $O(\log n)$ response time for INSERT(Q, a) and EXTRACT_MIN(Q). After all,

a software implementation on a sequential machine can guarantee $O(\log n)$ performance by using a height-balanced binary search tree. AVL trees, 2-3 trees, and B-trees are popular data structures which have this performance. For a sequential implementation of a single priority queue, a heap is an attractive data structure because heap storage can be managed as easily as stack storage. (Aho, Hopcroft, and Ullman [1974] has a good presentation of several of these techniques.) Unlike most programmed implementations of priority queues, however, the parallel structure required by the systolic multiqueue cannot use a separate data structure for each queue.

A major problem in the design of a hardware search tree is that the standard balanced tree schemes do not map well onto a fixed interconnection structure. A sequential algorithm can move the tree pointers to maintain the balance of the tree. Data usually remains in fixed locations. Since the "pointers" in a hardware tree are electrical wires, data must be moved to maintain the balance of the tree.

Because the systolic multiqueue requires the operations $\text{INSERT}(Q,a)$ and $\text{EXTRACT_MIN}(Q)$, keys are considered to be from a composite record $\langle Q,a \rangle$. A dummy queue number $+\infty$ is used to indicate an empty record. Records are compared by lexicographic ordering, that is, $\langle Q,a \rangle < \langle Q',a' \rangle$ if $Q < Q'$ or if $Q = Q'$ and $\text{key}(a) < \text{key}(a')$.

It is useful to view all operations on the tree as occurring in pairs $[\text{EXTRACT_MIN}(Q'), \text{INSERT}(Q,a)]$. Normally, the paired operation involves the dummy queue $+\infty$. For example, when an insertion is performed on an arbitrary queue, a $+\infty$ record is deleted by $\text{EXTRACT_MIN}(+\infty)$. If the systolic tree overflows from too many insertions, however, this exceptional condition can be handled by the operating system of the host computer. The job using a particular queue can be disabled and the elements in that queue can be removed. When an EXTRACT_MIN frees up some space, the elements of that queue can be "swapped" back in by the paired INSERT . The analogy to a virtual memory computer which has a swapping drum is a good one. Queues can be managed just like any other operating system resource. A small amount of bookkeeping is required to keep track of each queue, the number of items in the tree.

One scheme for implementing the paired operations is illustrated in Figure 5. Each processor in a systolic array is also a leaf of a systolic tree. A processor P_i contains one record $\langle Q_i, a_i \rangle$. The tree serves to broadcast paired operations to the processors and to retrieve the EXTRACT_MIN results from the processors. A paired operation $[\text{EXTRACT_MIN}(Q'), \text{INSERT}(Q,a)]$ will reach all the processors at the same time. Each

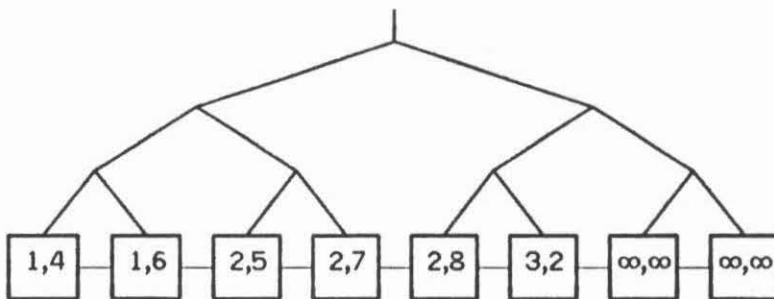


Figure 5: The systolic array-tree.

processor P_i executes:

1. Extraction. If $Q_i = Q'$ and $Q_{i-1} < Q'$, then send $\langle Q_i, a_i \rangle$ up the tree as the result of EXTRACT_MIN(Q').
2. Left shift. If $Q_{i-1} \geq Q'$, then shift $\langle Q_i, a_i \rangle$ left to P_{i-1} .
3. Right shift. If $\langle Q_i, a_i \rangle > \langle Q, a \rangle$, then shift $\langle Q_i, a_i \rangle$ right to P_{i+1} .
4. Insertion. If $\langle Q_{i-1}, a_{i-1} \leq \langle Q, a \rangle < \langle Q_i, a_i \rangle$, then P_i gets $\langle Q, a \rangle$.

During the first step, each processor checks to see whether it contains the item to be extracted. After that item is sent on its way up the tree, the elements to the right slide left to take up the empty slot. Then the position for the insertion is determined, and the elements to the right of that processor slide right to make room. Finally, the item to be inserted is placed in the slot left for it. Naturally, the shifts can be optimized so that those elements that slide both left and right do not actually have to move.

Whereas the array-tree keeps all the data at the leaves of the tree, the systolic tree shown in Figure 6 keeps the data in the internal nodes. Consequently, the structure is more like a standard search tree. The processor at each node holds two records, and has connections to its father and two sons. A depth-first tree traversal that prints the left record, recursively visits the left son, recursively visits the right son, and then prints the right record will print out the values in lexicographic order. There is a good reason for having the pointers between the records rather than the normal search tree method of a record between pointers. A balancing similar to the shift step in the systolic array-tree can be performed top-down to permit pipelining of the paired operations.

Each processor need only look at itself and its two sons to determine the shift. The topology of this tree is in some sense superior to the array-tree because there are fewer connections. This will be examined more closely in the next section.

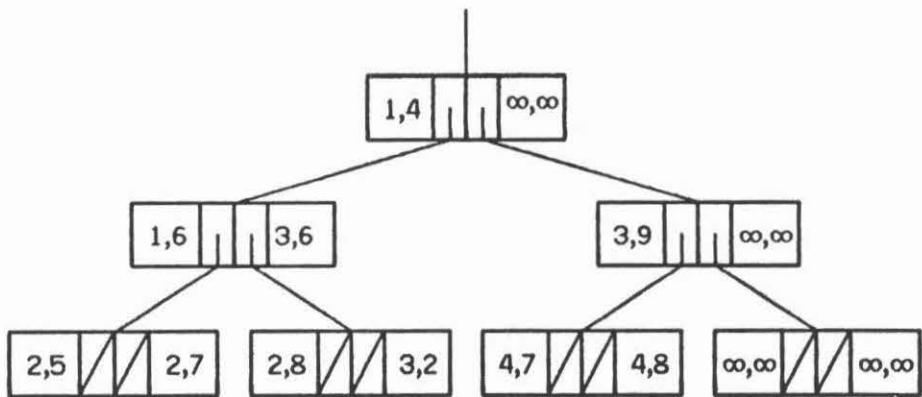


Figure 6: The systolic search tree.

6. VLSI Geometry of the Systolic Multiqueue

Simple and regular interconnections in a VLSI design lead to cheap implementations and high densities. Communication is costly in VLSI, and as the technology improves, the time and energy required for communication grows in comparison with that needed for processing. Therefore, the geometry of the systolic multiqueue must be considered in evaluating the cost of a VLSI implementation.

The linearly connected systolic array easily satisfies the requirement of having a simple geometric realization. The number of external data paths is small as well, emanating only from the ends of the structure. As was shown in Kung and Leiserson [1978], linearly connected systolic arrays are ideal for implementation in VLSI.

More interesting are the structures of the systolic binary trees. Mead and Rem [1978] substantiates the assumption that communication information from the leaf of a VLSI tree to the root takes time proportional to the height of the tree. The fact that the root is the only off-chip connection is highly desirable for VLSI where the number of pins on an IC package is a severe constraint.

The topology of a tree makes a planar embedding easy. In a technology where interconnections frequently occupy much of a chip, the simple connections of a binary tree leave more room for processing elements. It is easy to embed a binary tree in the plane using $O(n \log n)$ area for an n node tree. Figure 7 shows a geometry which realizes this bound. In fact, it is possible to embed a binary tree in the plane using area that is only linear in the number of nodes. This embedding is shown in Figure 8.

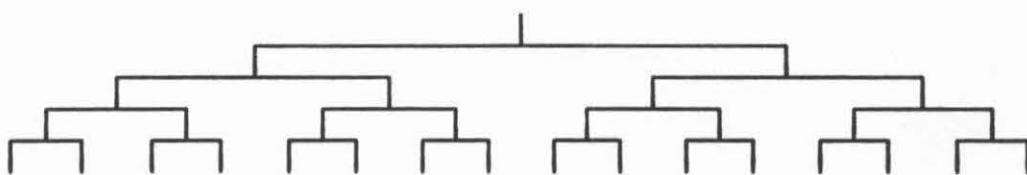


Figure 7: Embedding a binary tree in area $O(n \log n)$.

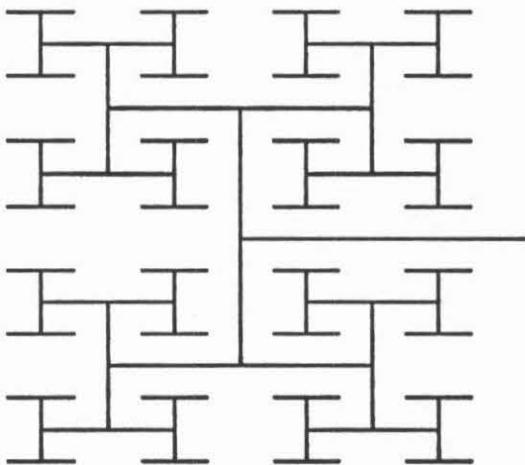


Figure 8: Embedding of a binary tree in linear area.

Whereas the systolic search tree presented in Section 5 can use either geometry, routing the linear connections in Figure 8 appears to be more complex. The tree part of the array-tree is used only for broadcasting, however, and a linear ordering of the leaves need not be the natural ordering shown in Figure 5. This makes the problem simpler, and leads to the linear area geometry of Figure 9.

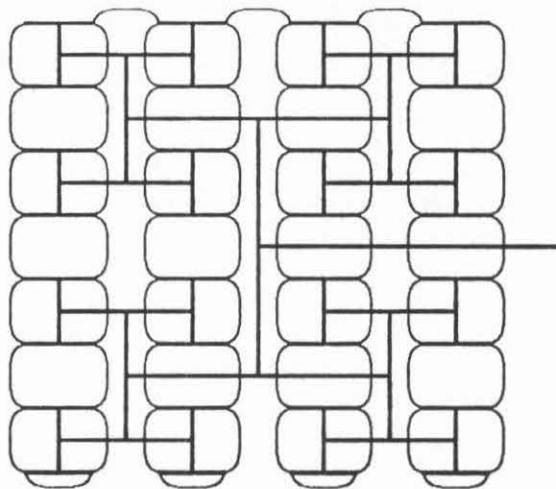


Figure 9: The systolic array-tree embedded in linear area.

There is an advantage to the geometry shown in Figure 7 over that in Figure 8. The linear area solution does not permit connections between the leaves of the tree and the edge of the chip. Although the systolic tree in the systolic multiqueue does not need connections to the leaves, the $O(n \log n)$ area embedding can be used to make a chip that will permit a larger systolic tree to be built up from several chips based on the linear area embedding. Figure 10 shows how this might be done. The decomposition can be very efficient since the number of linear area chips dwarfs the number of $O(n \log n)$ area chips.

7. Conclusion

The systolic multiqueue can be attached to a traditional computer system just like any other device. Because each operation on a queue takes constant time, however, it is

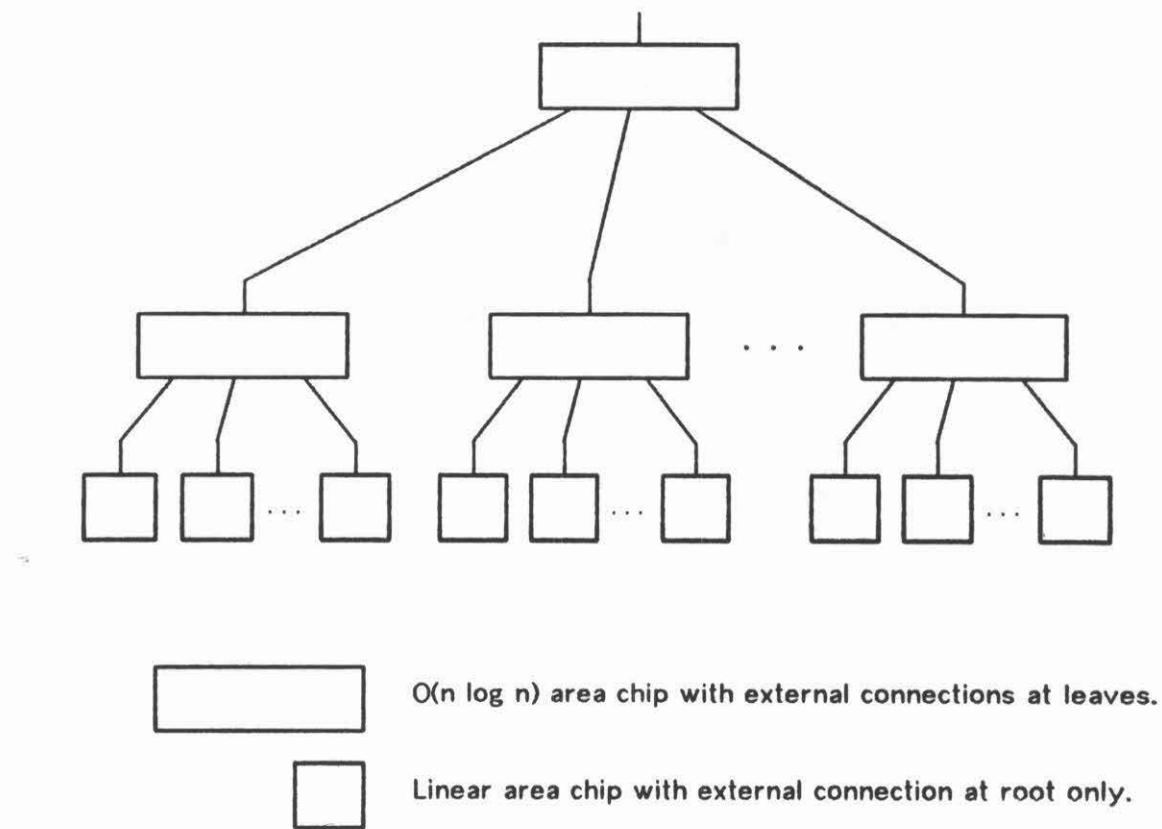


Figure 10: A large systolic tree as several VLSI chips.

reasonable to connect it directly to the CPU, and make the device visible to the user by extending the instruction set of the computer. Thus the INSERT operation might be a three operand instruction taking a queue number, a key, and a pointer to data. In a multiprogramming or timesharing environment there might be many users of the systolic multiqueue at the same time.

A priority queue is not an obscure data structure, and the uses of priority queues are many. The computation time of sorting alone is sufficient to justify the systolic multiqueue. Internal sorting normally takes $O(n \log n)$ time, but using the systolic multiqueue, we save a factor of $\log n$. Just insert the n items into one of the queues, and then execute n EXTRACT_MIN's. Not only does the computation take less time, but the

load on the CPU of the host machine is lessened. External sorts frequently use priority queues. For instance, the popular replacement selection algorithm (Knuth [1973], pp. 251-256) has a priority queue as its primary data structure.

Many types of search can be speeded up by utilizing fast priority queues. The A* algorithm (Nilsson [1971], pp. 57-65), for instance, chooses the best of many possible alternatives at each stage of the search. Many game playing programs using alpha-beta search sort the moves at each level in the game tree to increase the number of cut-offs. As the program searches deeper and deeper, the combinatorial explosion makes this very expensive, and therefore the sort is frequently abandoned at greater search depths. This need not be the case if the computer system has a systolic multiqueue.

In relational databases, the join operation is frequently implemented by sorting on the chosen fields of two relations and then performing a merge. Algorithms for finding the minimum spanning tree or convex hull of a set of points in a plane can use the systolic multiqueue. A priority queue is also useful for hidden line elimination. Priority queues are used in operating systems for resource management.

The systolic multiqueue provides insight into the organization of special-purpose multiprocessor devices with an emphasis on a VLSI implementation. The sharing of processors by several independent hardware structures is a key issue. A tree may not be the optimal shared structure for a given set of constraints. For example, a systolic array structure that performs like a Young tableau (Knuth [1973], pp. 48-72) might be better under certain conditions, although asymptotically the number of processors dedicated to a single queue grows as the square root of the number of shared processors rather than the logarithm.

The systolic multiqueue can be optimized and modified in many ways. For instance, it is easy to convert the systolic multiqueue into a systolic multideque that implements the priority deque operations INSERT, EXTRACT_MIN, and EXTRACT_MAX. Sten Andler has observed that because only one systolic array in the systolic multiqueue need operate at a time, the m systolic arrays of length $O(\log n)$ might be implemented using only $O(\log n)$ processors each having enough memory to hold m items. Various modifications can be made to the broadcast tree in the systolic array-tree and to the switching network in the systolic multiqueue.

Advances in microelectronics have made the realization of "smart" data structures a

practical reality. VLSI gives us the capability of building logic-in-memory hardware that will drastically change how things are computed. Models of computation based solely on the Von Neumann architecture will be insufficient to evaluate algorithms. Multiprocessor devices like the systolic multiqueue will introduce new cost functions to the sequential algorithm designer. But much work must be done to define and examine the models of parallel computation that lie between the mathematical world of computable functions and the physical world of space and time.

References

- Aho, Hopcroft, and Ullman [1974] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts.
- Knuth [1973] Knuth, Donald E., Sorting and Searching, Addison-Wesley, Reading, Massachusetts.
- Kung and Leiserson [1978] Kung, H. T. and Charles E. Leiserson, "Systolic Arrays (for VLSI)," in Mead and Conway [1978].
- Mead and Conway [1978] Mead, Carver A. and Lynn A. Conway, Introduction to VLSI Systems, to be published.
- Mead and Rem [1978] Mead, Carver A. and Martin Rem, "Highly Concurrent Structures with Global Communication," in Mead and Conway [1978].
- Nilsson [1971] Nilsson, Nils J., Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York.

Object Oriented Raster Displays

Bart Locanthy

California Institute of Technology

Pasadena, California 91125

This paper describes a special-purpose MOS/LSI memory device and its design. This device was designed as an attempt to speed up the display and animation of multicolor raster display images, and is highly specialized to that task. Generalizations of the approach taken would be suitable, and possibly economical for geometrical computations encountered in the course of manipulating integrated circuit layouts, such as design rule checking. These generalizations will be discussed, as will the design and implementation of this particular chip.

The principal motivation behind this design is the observation that the horizons of speed simply have not kept pace with the density improvements we have seen in semiconductor devices. These density improvements have brought about the personal computer, and along with it the re-emergence of interactive computer graphics. Indeed, the resolution and color range of bitmap displays is still increasing rapidly. However, the raw speed to manipulate these high resolution displays is not easy to come by. Witness the increasing development of 32 and 64 bit machines which have no reason to be that wide except to be able to blast display bits around faster. The objective of this design is to reduce the computer-to-display bandwidth required to make display changes at a given rate.

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency under contract number N00123-78-C-0806.

Figure 1 shows a typical bitmap display hierarchy with user interactions at the top and display interactions at the bottom. The user works at a high level of abstraction in a language (possibly graphical) tailored to his application. Users are typically low bandwidth devices. The computer interacts with the display in terms of bits in its memory, a very general form of interaction. The bandwidth available to change bits in memory is very high, although generally much less than is desired in terms of display speed.

Object Oriented Raster Displays

Although the computer eventually has to deal with bits, the data structure from which the bits must be generated is abstracted in varying degrees of specialization. With this specialization comes a more limited form of description and therefore more conciseness. Applications such as IC design often do not need the full generality of bitmap displays and can get by with this more limited form of description. With specialized LSI devices we can raise the primitive level of displayable objects and thus allow the computer to interact with the display in more abstract terms. The price is clear: generality. The payoff is conciseness and an effective increase in display bandwidth available.

The level of abstraction I have chosen to implement is that of rectangles. In a typical IC layout, the vast majority of displayed objects are rectangles or collections of rectangles. This is becoming increasingly true as DA systems attempt to do more and more for the user. Polygons and circular arcs are certainly more general, but the relative verbosity of expression weighed against the frequency of their use (not to mention complexity of implementation) favors the more simple representation of rectangles.

The processing for a bitmap display is amortized over a centralized display memory. Raster conversion is handled by the main processor, and display bits are read out either by hardware or special

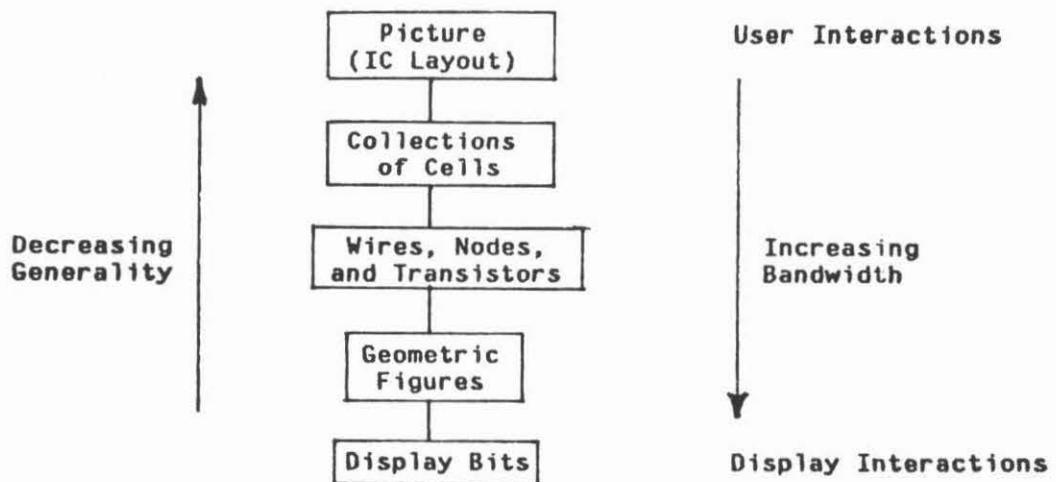


Figure 1. Typical Bitmap Display Hierarchy

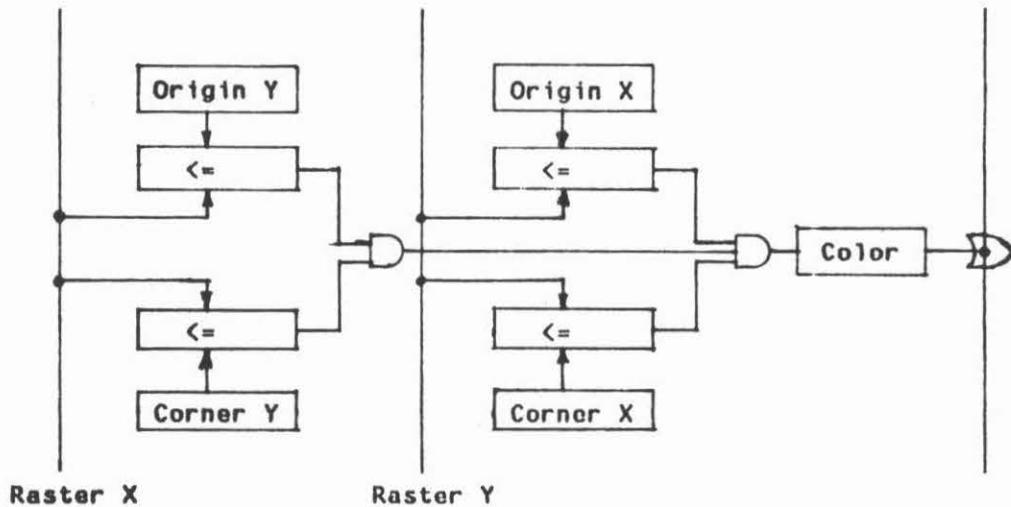


Figure 2. Rectangle Cell

microcode. No on-the-fly processing is performed. It is of course possible to speed up the raster conversion process while retaining the bitmap, resulting in an object oriented display of sorts. This approach was taken in the Xerox Alto, which has special microcode to manipulate rectangular areas in the bitmap. However, once the decision is made to store the display image in bitmap form, animation of the display becomes very tricky, since the images making up the display are ORed together. I think an adequate solution to this requires the display image to be regenerated for every frame. While it may be possible to contrive a combination of hardware and microcode to do this, I question how well this approach would scale for more complicated pictures or higher resolution displays.

A Rectangle Display Chip

A simpler solution is to distribute raster conversion hardware throughout the display memory. In addition to a high level description of its image, each object also knows how to convert itself to a raster image. For rectangles this raster conversion hardware is very simple indeed. Each rectangle only needs to be able to tell if it contains the raster point, and give its contribution to the color if this is so. The raster scans through the entire screen and the color at each point is determined by combining the contributions from each rectangle.

Figure 2 shows the organization of a rectangle cell. For convenience, the boundaries of the rectangle are given in absolute coordinates and are determined by two diagonally opposing vertices, call them the origin and corner. Each rectangle has a color, which is conveniently represented as an address in a color map. Colors are combined by ORing color map addresses. The test for containment is simple (i.e. $\text{contains}(p) = \text{origin} \leq p \text{ AND } \text{corner} \geq p$), and minimally obtained by ANDing the carries out of four subtractions. The process of raster conversion is then broadcasting the raster coordinates to all rectangle objects and ORing the color map address to produce the appropriate color for each point.

Figure 3 is a schematic for one bit of a register/comparator pair. The storage element is a standard 6 transistor static RAM cell. The comparator is a simple alternating polarity ripple through design using 7 transistors. As figure 4 demonstrates, the space occupied by logic almost exactly equals the space occupied by storage. This is a good balance of memory and logic for a static organization. However, the use of dynamic storage devices would upset this balance considerably, since the area required for logic would not decrease appreciably. Even with extreme specialization and application of cleverness, the fact remains that logic is expensive. (Similar arguments make the construction of associative memories hard to justify.) Imagine if you will the gross imbalance that would result from attempting a more general primitive element than rectangles.

Cavalier application of conservative design rules (6 micron single level polysilicon, no buried contacts) resulted in 8 rectangles on a chip which measures about 5000 microns square. Using 8 bit precision in x and y coordinates and color, this amounts to 320 bits of storage. In a chip where half of the interior area is occupied by logic, this gives some indication of the state of integrated circuit technology available to a university. A technology capable of producing a 16K static RAM could produce an 8K bit rectangle chip. This translates to 200 rectangles of 8 bit precision, or 125 rectangles of 12 bit precision.

Scaling

Object oriented displays scale in a particularly simple way. The amount of storage required for application A is proportional to the number of objects displayed in application A. Capability can be added simply by adding more storage. Higher precision displays are generated by using higher resolution parts. Bitmap displays, however, require memory capacity AND bandwidth proportional to the square of the precision desired.

No claims are made for the coding density of object oriented displays. Naturally, simple pictures require very few parts.

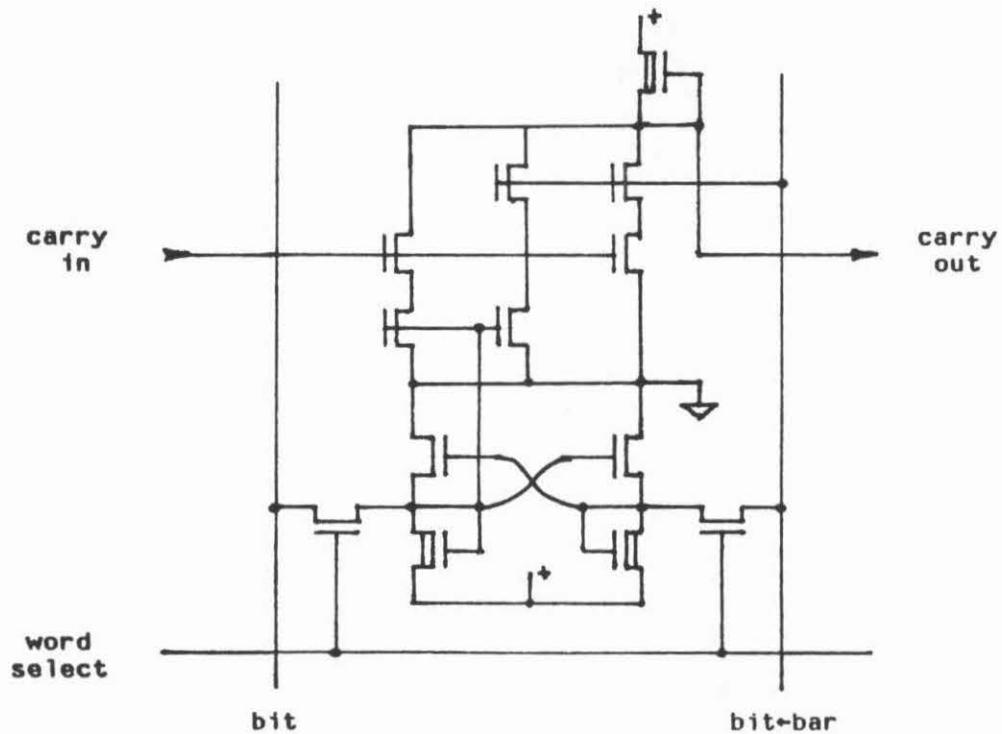


Figure 3. Storage/Comparator Schematic

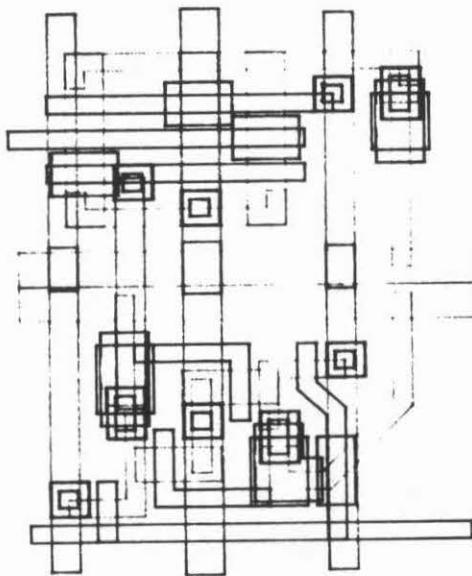


Figure 4. Storage/Comparator Layout

Conversely, complicated pictures may require more bits of object oriented storage than bitmap storage. The object oriented display is a run length, or derivative encoding of a picture. No particular relation exists for the amount of storage needed to express the spatial derivative of a picture as compared to that needed for the picture itself, although previous work with run length encoding of pictures indicates that these amounts are usually about the same.

Certain phases of integrated circuit design involve the display of fairly simple pictures with which a high degree of interaction is desired. During other phases it is desirable to see large fractions of the circuit at the same time. This can mean the display of literally hundreds of thousands of rectangles for a typical VLSI chip, each rectangle only occupying a few pixels. For these cases it would be desirable to have a bitmap storage backup for an object oriented display, using the object storage as a fast raster conversion hardware front end.

Speed

Each element of an object oriented display must perform a computation for every pixel in every frame. Some high resolution non-interlaced displays have bit times on the order of 15 nanoseconds. Even taking advantage of the simplicity of rectangles, there is still some question as to whether the processors will be able to keep up with the raster or not. Let me amend that. For a 15 nanosecond per pixel display, there is no question at all.

The ripple carry subtraction circuit used in the rectangle display chip simulated to about 10 nanoseconds per bit. An 8 bit compare would thus require 80 nanoseconds in the worst case, the worst case being at the boundary of a rectangle. This is certainly sufficient for the 256 by 256 resolution possible with an 8 bit chip. For higher resolutions, the edge detection decisions will always arrive late, and accuracy will suffer from the variation in speed of the rectangle processors.

This variation can be equalized by clocking the individual carries

and skewing the bits of the raster coordinates as they are presented to the rectangle processors. Skewing and pipelining carries is also a convenient way of building high speed counters, which are needed to calculate the raster position and should be placed on-chip anyway. It seems therefore that a very high speed object oriented storage chip could be constructed without much difficulty.

Another approach to the speed problem is to keep a completely static design and let the logic be as fast as it turns out to be. Regardless of how slow the comparison logic turns out to be, the worst that can happen is that it can't keep up with changes in the low order raster bits. We are guaranteed monotonicity, only linearity suffers.

Regardless of whether we believe improvements in IC processes will eventually be able to cope with the speed or not, it makes sense to adopt the simplest possible design. A simple design will be compact and will thus enjoy some advantage in speed over a more complicated part. It may also work, an important consideration in a world where turnaround is measured in months.

System Integration

Where does an object oriented storage device fit into a system? Figure 5 illustrates a suitable embellishment, Class Rectangle, transcribed into Simula from a successor object oriented language, Smalltalk. By the use of this mechanism, the user deals with the protocol of the general rectangle object. The fact that certain operations such as display and movement are implemented in hardware is invisible. All the user cares is that the rectangles he creates do his bidding; Class Rectangle handles the details of interacting with the display memory.

Certain operations of Class Rectangle are not handled directly by the rectangle chip, but are maddeningly close to the basic function. The test for containment of a point is a basic primitive of each rectangle processor, yet it is not conveniently available

```
CLASS rectangle(origin,corner,color);
REF(point) origin,corner; INTEGER color;
! Excerpts from SSP's Class rectangle;
BEGIN

    REF(point) PROCEDURE center;
    center :- origin.plus(corner).times(0.5);

    PROCEDURE show;
    ! This is what it would be for a bitmap display;
    BEGIN
        INTEGER i,j;
        FOR i := origin.x TO corner.x DO
            FOR j := origin.y TO corner.y DO
                displaycolor(i,j) := displaycolor(i,j) OR color;
    END;

    PROCEDURE moveby(p); REF(point) p;
    ! includes an implicit show;
    BEGIN
        origin :- origin.plus(p);
        corner :- corner.plus(p);
    END;

    BOOLEAN PROCEDURE contains(p); REF(point) p;
    contains := origin <= p AND corner >= p;

    REF(rectangle) PROCEDURE clip(r); REF(rectangle) r;
    clip :- IF origin > r.corner OR corner < r.origin THEN NONE
    ELSE NEW rectangle(origin.max(r.origin),corner.min(r.corner));

    REF(rectangle) PROCEDURE including(p); REF(point)p;
    ! returns the rectangle including the point p;
    including:-NEW rectangle(origin.min(p),corner.max(p));

END;
```

Figure 5. Class Rectangle

from outside, except by using the color field of each rectangle as an address and priority encoding the colors out from the assorted chips. Containment when applied to a set of rectangles gives solves the pointing problem in constant time.

Even more useful would be the test for overlap applied to a set of rectangles. The basic test for overlap involves two point comparisons, each of which is similar to the comparisons performed by the rectangle chip. If this operation were available from the outside, the mutual overlap of a set of rectangles could be found in linear time. Whether or not this is useful to you depends on how much automatic design rule checking you like to do. Anticipating the return of some rectangle display chips for testing, I may soon find out just how much automatic design rule checking I should have done.

Summary

A highly specialized MOS/LSI integrated circuit has been described. The sole reason for the existence of the circuit is to speed the display and animation of multicolor raster display images composed of rectangles. By such specialization it is hoped that the circuit will perform it's intended task faster and more economically than bitmap displays.

Using standard silicon gate technology and relaxed design rules, the circuit provides storage and raster conversion hardware for 8 rectangles, using 8 bit precision, on a chip measuring about 5000 microns square. While generalizations are possible and in some cases useful, I felt the added cost in design time and circuit complexity could not be justified for the first implementation.

There are of course abstract pleasures from designing systems cleanly in an object oriented way. However, the ultimate justification for me will come when I can drag parts of an IC layout across a display that comes alive with the animation made possible by this kind of "smart" memory device.

References

1. Birtwistle, G., Dahl, O. J. Dahl et al, Simula Begin, Petrocelli/ Charter 1973.
2. Ingalls, D., The Smalltalk-76 Programming System: Design and Implementation, Proceedings of the ACM Conference on Principles of Programming Languages, February 1978.

Storage Management in a LISP-based Microprocessor

Guy Lewis Steele Jr.* and Gerald Jay Sussman**

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

* Fannie and John Hertz Fellow

** Esther and Harold E. Edgerton Associate Professor of Electrical Engineering

Abstract

We present a design for a class of computers whose "instruction sets" are based on LISP. LISP, like traditional stored-program machine languages and unlike most high-level languages, conceptually stores programs and data in the same way and explicitly allows programs to be manipulated as data. LISP is therefore a suitable language around which to design a stored-program computer architecture. LISP differs from traditional machine languages in that the program/data storage is conceptually an unordered set of linked record structures of various sizes, rather than an ordered, indexable vector of integers or bit fields of fixed size. The record structures can be organized into trees or graphs. An instruction set can be designed for programs expressed as such trees. A processor can interpret these trees in a recursive fashion, and provide automatic storage management for the record structures.

We concentrate here on the issues of memory management in such a computer, and the reasons why a layered design strategy is not only desirable and natural but even mandatory.

A prototype VLSI LISP microprocessor has been designed and fabricated for testing. It is a small-scale version of the ideas presented here, containing a sufficiently complete instruction interpreter to execute small programs, and a rudimentary storage allocator. We intend to design and fabricate a full-scale VLSI version of this architecture in 1979.

Keywords: microprocessors, large scale integration, integrated circuits, VLSI, LISP, SCHEME, list structure, garbage collection, storage management.

Introduction

An idea which has increasingly gained attention is that computer architectures should reflect specific language structures to be supported. This is an old idea; one can see features in the machines of the 1960's intended to support COBOL, FORTRAN, ALGOL, and PL/I. More recently research has been conducted into architectures to support string or array processing as in SNOBOL or APL.

An older and by now well-accepted idea is that of the stored-program computer. In such a computer the program and the data reside in the same memory; that is, the program is itself data which can be manipulated as any other data by the processor. It is this idea which allows the implementation of such powerful and incestuous software as program editors, compilers, interpreters, linking loaders, debugging systems, etc.

One of the great failings of most high-level languages is that they have abandoned this idea. It is extremely difficult, for example, for a PL/I (PASCAL, FORTRAN, COBOL ...) program to manipulate PL/I (PASCAL, FORTRAN, COBOL ...) programs.

On the other hand, many of these high-level languages have introduced other powerful ideas not present in standard machine languages. Among these are (1) recursively defined, nested data structures; and (2) the use of functional composition to allow programs to contain expressions as well as (or instead of) statements. The LISP language in fact has both of these features. It is unusual among high-level languages in that it also explicitly supports the stored-program idea: LISP programs are represented in a standardized way as recursively defined, nested LISP data structures. By contrast with some APL implementations, for example, which allow programs to be represented as arrays of characters, LISP also reflects the structure of program expressions in the structure of the data which represents the program. (An array of APL characters must be parsed to determine the logical structure of the APL expressions represented by the array. Similar remarks apply to SNOBOL statements represented as SNOBOL strings.)

It is for this reason that LISP is often referred to as a "high-level machine language". As with standard stored-program machine languages, programs and data are made of the same stuff. In a standard machine, however, the "stuff" is a linear (ordered) vector of fixed-size bit fields; a program is represented as an ordered sequence of bit fields (instructions) within the overall vector. In LISP, the "stuff" is a heterogenous (unordered) set of records of various sizes linked to form lists, trees, and graphs; a program is represented as a tree (a "parse tree" or "expression tree") of linked records (a subset of the overall set of records). Standard machines usually exploit the linear nature of the "stuff" through such mechanisms as indexing by additive offset and linearly advancing program counters. A computer based on LISP can similarly exploit tree structures. The counterpart of indexing is component selection; the counterpart of linear instruction execution is evaluation of expressions by recursive tree-walk.

Just as the "linear vector" stored-program-computer model leads to a variety of specific architectures, so with the "linked record" model. For concreteness we present here one specific architecture based on the linked record model which has actually been constructed.

List Structure and Programs

One of the central ideas of the LISP language is that storage management should be completely invisible to the programmer, so that he need not concern himself with the issues involved. LISP is an object-oriented language, rather than a value-oriented language. The LISP programmer does not think of variables as the objects of interest, bins in which values can be held. Instead, each data item is itself an object, which can be examined and modified, and which has an identity independent of the variable(s) used to name it.

The precise form of LISP data objects is not of concern here. The most important one, however, is the list cell (or cons cell), which is a record with two components which are pointers to other data objects; such cells are chained together by their pointers to form arbitrarily complicated graph structures. LISP provides primitive operators (CAR and CDR) following pointers, and more complex operators (ASSOC, MEMBER, PUTPROP, GET, etc.) for searching and restructuring such graphs in stylized ways.

The philosophically most important operator (CONS) effectively creates a new list cell "out of thin air". As far as the LISP programmer is concerned, new data objects are available in endless supply. They can be conveniently called forth to serve some immediate purpose and discarded when they are no longer of use. While creation is explicit, discarding is not; a data object simply disappears into limbo when the program throws away all references (direct or indirect) to that object.

The immense freedom this gives the programmer may be seen by an example taken from current experience. A sort of error message familiar to most programmers is "too many nested DO loops" or "more than 200 declared arrays" or "symbol table overflow". Such messages typically arise within compilers or assemblers which were written in languages requiring data tables to be pre-allocated to some fixed length. The author of a compiler, for example, might well guess, "No one will ever use more than, say, ten nested DO loops; I'll double that for good measure, and make the nested-DO-loop-table 20 long." Inevitably, someone eventually finds some reason to write 21 nested DO loops, and finds that the compiler overflows its fixed table and issues an error message (or, worse yet, doesn't issue an error message!). On the other hand, had the compiler writer made the table 100 long or 1000 long, most of the time most of the memory space devoted to that table would be wasted.

A compiler written in LISP would be much more likely to keep a linked list of records describing each DO loop. Such a list could be grown at any time by creating a new record on demand and adding it to the list. In this way as many or as few records as needed could be accommodated.

Now one could certainly write a compiler in any language and provide such dynamic storage management with enough programming. The point is that LISP provides automatic storage management from the outset and encourages its use (in much the same way that FORTRAN provides floating-point numbers and encourages their use, even though the particular processor on which a FORTRAN program runs may or may not have floating-point hardware).

Besides providing automatic storage management, LISP provides a standardized representation for programs using standard data structures within the language. Moreover, it provides a standard operator (EVAL) which will interpret a program expressed in this standard form. This operator can itself be expressed in LISP, and the definition of such an operator constitutes a description of a processor which can execute LISP programs. This operator must be able to traverse the data structures representing the program being executed; and, because of the recursive nature of the interpretation process, needs additional temporary storage to hold intermediate results and control information. This storage can be in the form of LISP data objects.

A complete LISP system can therefore be conveniently divided into two parts: (1) a storage system, which provides an operator for the creation of new data objects and also other operators (such as pointer traversal) on those objects; and (2) a program interpreter, which executes programs expressed as data structures within the storage system. (Note that this memory/processor division characterizes the usual von Neumann architecture also. The differences occur in the nature of the processor and the memory system.)

Storage Management

Most hardware memory systems which are currently available commercially are not organized as sets of linked lists, but rather as the usual linearly-indexed vectors. (More precisely, commercially available RAMs are organized as Boolean N-cubes indexed by bit vectors. The usual practice is to impose a total ordering on the memory cells by ordering their addresses lexicographically, and then to exploit this total ordering by using indexing hardware.)

Commercially available memories are, moreover, available only in finite sizes (more's the pity). Now the free and wasteful throw-away use of data objects would cause no problem if infinite memory were available, but within a finite memory it is an ecological disaster. In order to make such memories useable to our processor we must interpose a storage manager which makes a finite vector memory appear to the evaluation mechanism to be an infinite linked-record memory. This would seem impossible, and it is; the catch is that at no time may more records be active than will fit into the finite memory actually provided. The memory is "apparently infinite" in the sense that an indefinitely large number of new records can be "created". The storage manager recycles discarded records in order to create new ones in a manner completely invisible to the LISP program interpreter.

The microprocessor we have designed therefore actually consists of two processors, side by side. One (the evaluator, EVAL) operates in terms of a LISP-style record-structure memory, and interprets LISP programs. The other (the storage manager, or garbage collector (as it is traditionally called in LISP systems), GC) serves as an intermediary between EVAL and the external memory. GC deals with the finiteness of the external memory by locating data objects which have been discarded and making them available for recycling. GC also provides EVAL with operators for dealing with the data objects.

The storage representation is a standard one using two consecutive words of memory to hold a list cell, where each word of memory can hold a pointer consisting of a type field and an address field. The address part of a pointer is in turn the address within the linear memory of the record pointed to. (This may seem obvious, but remember that until now we have been noncommittal about the precise representation of pointers, as until this point all that was necessary was that the memory system associate records with pointers by any convenient means whatsoever. The evaluator is completely unconcerned with the format or meaning of addresses; it merely accepts them from the memory system and eventually gives them back later to retrieve record components. One may think of an address as a capability for accessing a record using certain defined operations such as "fetch component number 1".)

New data objects are created in successively higher memory locations. When the finite memory is exhausted, the storage manager performs a garbage collection procedure which determines which data objects are accessible to the evaluator and which not, and compacts the former toward the low end of the memory, leaving the unused memory space in a block at the high end in which to allocate new objects.

Many techniques for garbage collection are well-documented in the literature [McCarthy 1962] [Minsky 1963] [Hart 1964] [Saunders 1964] [Schorr 1967] [Conrad 1974] [Baker 1978] [Morris 1978], and will not be discussed here. (In fact, the storage manager in the prototype we have constructed actually includes no garbage collector. The prototype was one project of a "project set" including some two dozen separate circuits, all of which had to be fit onto a single chip together. This imposed severe area limitations which restricted the address size to eight bits, and required the elimination of the microcode for the garbage collector. We anticipate no obstacles to including a garbage collector in a full-sized single-chip processor. The complexity of a simple garbage collector is comparable to that of the evaluator shown above.)

Physical Layout and Implementation

The evaluator and the storage manager are each implemented in the same way as an individual processor. Each processor has a state-machine controller and a set of registers. On each clock cycle the state-machine outputs control signals for the registers and also makes a transition to a new state.

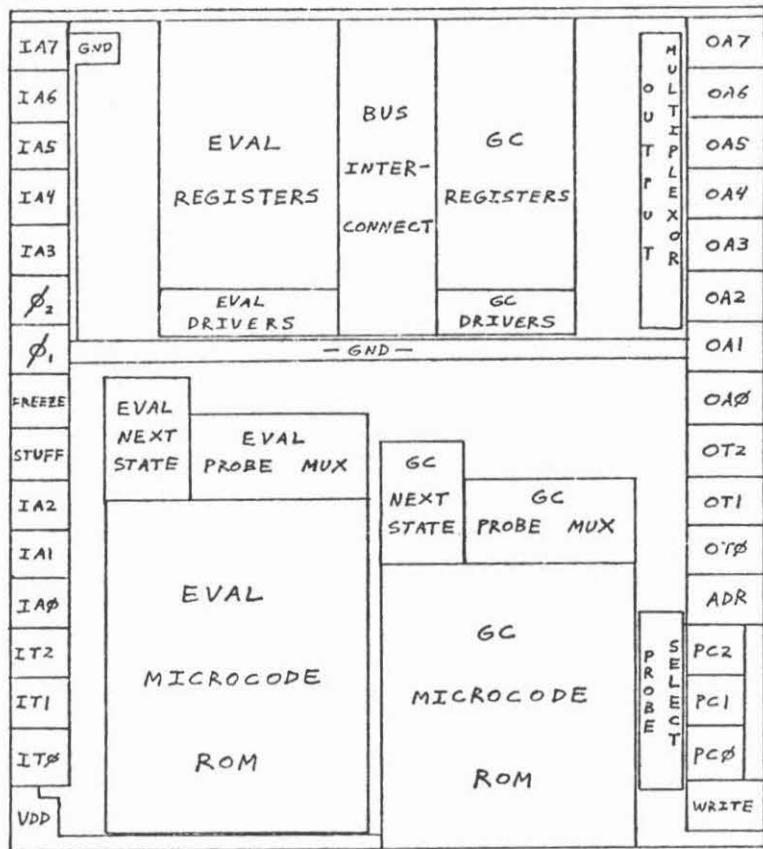
The contents of any register is a pointer, containing an address field (8 bits in the prototype) and a type field (3 bits in the prototype). The registers of a processor are connected by a common bus (E bus in the evaluator, G bus in the storage manager). Signals from the controller can read at most one register onto the bus, and load one or more other registers from the bus. One register in each controller has associated incrementation logic; the controller can cause the contents of that register, with 1 added to its address part, to be read onto the bus. The controller can also force certain constant values onto the bus rather than reading a register.

The processors can communicate with each other by causing the E and G busses to be connected. The address and type parts of the busses can be connected separately. (Typically the E bus might have its address part driven from the G bus and its type part driven by a constant supplied by the evaluator controller.) The G bus can also be connected to the address/data lines for the off-chip memory system. The storage-manager controller produces additional signals (ADR and WRITE) to control the external memory. In a similar manner, the evaluator controller produces signals which control the storage manager. (Remember that from the point of view of the evaluator, the storage manager is the memory interface!)

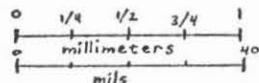
Each controller effectively has an extra "state register" which may be thought of as its "micro-PC". At each step the next state is computed by combining its current state with external signals in the following manner. Each "microinstruction" has a field explicitly specifying the next desired state, as well as bits specifying possible modifications of that state. If specified, external signals are logically OR'd into the desired state number. In the prototype evaluator these external signals are: (1) the type bits from the E bus; (2) a bit which is 1 iff the E bus type field is zero and a bit which is 1 iff the E bus address is zero. In the storage manager these signals are: (1) the four control bits from the evaluator controller; (2) a bit which is 1 iff the G bus address is zero. This is the way in which dispatching is achieved.

Once this new state is computed, it is passed through a three-way selector before entering the state register. The other two inputs to the selector are the current state and the data lines from the external memory system. In this way the selector control can "freeze" a controller in its current state by recirculating it, or jam an externally supplied state into the state register (both useful for debugging operations). The "freeze" mechanism is used by the storage manager to suspend the evaluator until it is ready to process the next request. In the same way, the external memory can suspend the storage manager by asserting the external FREEZE signal, thereby causing a "wait state".

(The FREEZE signal is provided as a separate control because the dynamic logic techniques usual in NMOS were used; if one stopped the processor simply by stopping the clock, the register contents would dissipate. The clocks must keep cycling in order to "refresh" the registers. The state recirculation control allows the machine to be logically stopped despite the fact that data is still circulating internally. We discovered that this technique imposed constraints on other parts of the design: the incrementation logic is the best example. It was originally intended to design an incrementing counter register, which would increment its contents in place during the recirculation of a clock cycle in which an "increment" signal was asserted. If this had been done, however, and the processor were frozen during an instruction which asserted this signal, the counter would continue to count while the processor was stopped! This could have been patched by having the FREEZE signal override the increment signal, but it was deemed simpler to adopt a design strategy in which nothing at the microcode level called for any data to be read, modified, and stored back into the same place. Thus in the actual design one must read data through modification logic and



Physical layout of
prototype LISP processor



3.96 mm x 3.38 mm

156 mils x 133 mils

then onto the bus, to be stored in a different register; then if this operation is repeated many times because of the FREEZE signal it makes no difference.)

Each state-machine controller consists of a read-only memory (implemented as a programmed-logic-array), two half-registers (clocked inverters, one at each input and one at each output), and some random logic (e.g. for computing the next state). The controllers are driven by externally supplied two-phase non-overlapping clock signals; on phase 1 the registers are clocked and the next state is computed, and on phase 2 the next-state signals appear and are latched.

All of the signals from the two controllers ($62 = 34+28$ in the prototype) are multiplexed onto twelve probe lines by six unary probe-control signals. (These signals are derived from the three binary-encoded off-chip signals PC0-PC2.) When a probe-control signal is asserted, the memory output pads (11 data pads plus the ADR signal in the prototype) are disconnected from the G bus and connected to the twelve probe lines. In this way the chip can be frozen and then all controller outputs verified (by cycling the probe-control signals through all six states). Also recall that the controller states can be jammed into the state registers from the memory input pads. This should allow the controller microcode to be tested completely without depending on the registers and busses working.

The diagram shows the physical layout of the prototype chip. The two controllers are side by side, with the evaluator on the left and the storage manager on the right. Above each controller is the next-state logic and probe multiplexor for that controller. Above those are the register arrays, with the busses running horizontally through them. The bus connections are in the center. The input pads are on the left edge, and the output pads on the right edge. The input pads are bussed through the evaluator's register array parallel to the E bus lines, so that they can connect to the G bus. (Unfortunately, there was no time to design tri-state pads for this project.)

Discussion

A perhaps mildly astonishing feature of this computer is that it contains no arithmetic-logic unit. More precisely, it does have arithmetic and logical capabilities, but the arithmetic units can only add 1, and the logical units can only test for zero. (Logicians know that this suffices to build a "three-counter machine", which is known to be as universal (and as convenient!) as a Turing Machine. However, our LISP architecture is also universal, and considerably more convenient.)

LISP itself is so simple that the interpreter needs no arithmetic to run interesting programs (such as computing symbolic derivatives and integrals, or pattern matching). All the LISP interpreter has to do is shuffle pointers to and from memory, and occasionally dispatch on the type of a pointer. The incrementation logic is included on the chip for two reasons. In the evaluator it is used for counting down a list when looking up lexical variables in the environment; this is not really necessary, for there are alternative environment representation strategies. In the storage manager incrementation is necessary (and, in the prototype, sufficient) for imposing a

total ordering on the external memory, so as to be able to enumerate all possible addresses. The only reason for adding 1 is to get to the next memory address. (One might note that the arithmetic properties of general two-argument addition are not exploited here. Any bijective mapping from the set of external memory addresses onto itself (i.e. a permutation function) would work just fine (but the permutation should contain only one cycle if memory is not to be wasted!). For example, subtracting 1 instead of adding, or Gray-code incrementation, would do.)

This is not to say that real LISP programs do not ever use arithmetic. It is just that the LISP interpreter itself does not require binary arithmetic of the usual sort. This architecture is intended to use devices which are addressed as memory, in the same manner used by the PDP-11, for example. We envision having a set of devices on the external memory bus which do arithmetic. One would then write operands into specific "memory locations" and then read arithmetic results from others. Such devices could be very complex processors in themselves, such as specialized array or string processors. In this way the LISP computer could serve as a convenient controller for other processors, for one thing LISP does well is to provide recursive control and environment handling without much prejudice (or expertise!) as to the data being operated upon.

Expanding on this idea, one could arrange for additional signals to the external memory system from the storage manager, such as "this data item is needed (or not needed)", which would enable external processors to do their own storage management cooperatively with the LISP processor. One might imagine, for example, an APL machine which provided tremendous array processing power, controlled by a LISP interpreter specifying which operations to perform. The APL machine could manage its own array storage, using a relatively simple storage manager cued by "mark" signals from the LISP storage manager.

The possibility of additional processors aside, this architecture exhibits an interesting layered approach to machine design. One can draw boundaries at various places such that everything above the boundary is a processor which treats everything below the boundary as a memory system with certain operations. If the boundary is drawn between the evaluator and the storage manager, then everything below the boundary together constitutes a list-structure memory system. If it is drawn between the storage manager and the external memory, then everything below the boundary is the external memory. Supposing the external memory to be a cached virtual memory system, then we could draw boundaries between the cache and main memory, or between main memory and disks, and the same observation would hold. At the other end of the scale, a complex data base management system could be written in LISP, and then the entire LISP chip (plus some software, perhaps in an external ROM) would constitute a memory system for a data base query language interpreter. In this manner we have a layered series of processors, each of which provides a more sophisticated memory system to the processor above it in terms of the less sophisticated memory system below it.

Another way to say this is that we have a hierarchy of data abstractions, each implemented in terms of a more primitive one. Thus the storage manager makes a finite, linear memory look "infinite" and tree-structured. A cache system makes a large, slow memory plus a small, fast memory look like a large, fast memory.

Yet another way to view this is as a hierarchy of interpreters running in virtual machines. Each layer implements a virtual machine within which the next processor up operates.

It is important to note that we may choose any boundary and then build everything below it in hardware and everything above it in software. Our LISP system is actually quite similar to those before it, except that we have pulled the hardware boundary much higher. One can also put different layers on different chips (as with the LISP chip and its memory). We choose to put the evaluator and the storage manager on the same chip only because (a) they fit, and (b) in the planned full-scale version, the storage manager would need too many pins as a separate chip.

Each of the layers in this architecture has much the same organization: it is divided into a controller ("state machine") and a data base ("registers"). There is a reason for this. Each layer implements a memory system, and so has state; this state is contained in the data base (which may be simply a small set of references into the next memory system down). Each layer also accepts commands from the layer above it, and transforms them into commands for the layer below it; this is the task of the controller.

We have already mentioned some of the analogies between a LISP-based processor and a traditional processor. Corresponding to indexing there is component selection; corresponding to a linearly advancing program counter there is recursive tree-walk of expressions. Another analogy we might draw is to view the instruction set as consisting of variable-length instructions (whose pieces are joined by pointers rather than being arranged in sequential memory locations). Each instruction (variable reference, call to CONS, call to use function, etc.) takes a number of operands. We may loosely say that there are two addressing modes in this architecture, one being immediate data (as in a variable reference), and the other being a recursive evaluation. In the latter case, merely referring to an operand automatically calls for the execution of an entire routine to compute it!

Project History

In January 1978 one of us (Sussman) attended a course given at MIT by Charles Botchek about the problems of integrated circuit design. There he saw pictures of processors such as 8080's which showed that half of the chip area was devoted to arithmetic and logical operations and associated data paths. On the basis of our previous work on LISP and SCHEME [Sussman 1975] [Steele 1976a] [Steele 1976b] [Steele 1977] [Steele 1978a] [Steele 1978b] it occurred to him that LISP was sufficiently simple that almost all the operations performed in a LISP interpreter are dispatches and register shuffles, and require almost no arithmetic. He concluded that if you could get rid of the ALU in a microprocessor, there would be plenty of room for a garbage collector, and one could thus get an entire LISP system onto a chip. He also realized that typed pointers could be treated as instructions, with the types treated as "opcodes" to be dispatched on by a state machine. (The idea of typed pointers came from many previous implementations of LISP-like languages, such as MUDDLE [Galley 1975], ECL [Wegbreit 1974], and the LISP

Machine [Greenblatt 1974]. However, none of these uses the types as opcodes in the evaluator. This idea stemmed from an aborted experiment in nonstandard LISP compiler design which we performed in 1976.)

In the summer of 1978 Sussman wrote a LISP interpreter based on the state machine specification. It worked.

In the fall of 1978 Lynn Conway came to MIT from Xerox PARC as a visiting professor to teach a subject (i.e. course) on VLSI design which she developed with Carver Mead of Caltech. Sussman suggested that Steele take the course "because it would be good for him" (and also because he couldn't sit in himself because of his own teaching duties). Steele decided that it might be interesting. So why not?

The course dealt with the structured design of NMOS circuits. As part of the course each student was to prepare a small project, either individually or collaboratively. (This turned out to be a great success. Some two dozen projects were submitted, and nineteen were fit together onto a single 7 mm x 10 mm project chip for fabrication by an outside semiconductor manufacturer and eventual testing by the students.)

Now Steele remembered that Sussman had claimed that a LISP processor on a chip would be simple. A scaled-down version seemed appropriate to design for a class project. Early estimates indicated that the project would occupy 2.7 mm x 3.7 mm, which would be a little large but acceptable. (The average student project was a little under 2 mm x 2 mm.) The LISP processor prototype project would have a highly regular structure, based on programmed logic array cells provided in a library as part of the course, and on a simple register cell which could be replicated. Hence the project looked feasible. Steele began the design on November 1, 1978.

The various register cells and other regular components took about a week to design. Another week was spent writing some support software in LISP, including a microassembler for the microcode PLAs; software to produce iterated structures automatically, and rotate and scale them; and an attempt to write a logic simulator (which was "completed", but never debugged, and was abandoned after three days).

The last three weeks were spent doing random interconnect of PLA's to registers and registers to pads. The main obstacle was that there was no design support software for the course other than some plotting routines. All projects had to be manually digitized and the numbers typed into computer files by keyboard (the digitization language was the Caltech Intermediate Format (CIF)). This was rather time-consuming for all the students involved.

In all the design, layout, manual digitization, and computer data entry for this project took one person (Steele) five weeks of full-time work spanning five and one-half weeks (with Thanksgiving off). This does not include the design of the precise instruction set to be used, which was done in the last week of October (and later changed!). (The typical student project also took five weeks, but presumably with somewhat less than full-time effort.)

During this time some changes to the design were made to keep the area down, for as the work progressed the parts inexorably grew by 20 microns here and 10 microns there. The number of address bits was chopped from ten to eight. A piece of logic to compare two addresses for equality (to implement the LISP EQ operation) was scrapped (this logic was to provide an additional

dispatch bit to the evaluator in the same group as the E-bus-type-zero bit and the E-bus-address-zero bit). The input pad cell provided in the library had to be redesigned to save 102 microns on width. The WRITE pad was connected to the bottom of the PLA because there was no room to route it to the top, which changed the clock phase on which the WRITE signal rose, which was compensated for by rewriting the microcode on the day the project was due (December 6, 1978). Despite these changes, the area nevertheless increased. The final design occupied 3.378 mm x 3.960 mm.

The prototype processor layout file was merged with the files for the other students' projects, and the project chip was sent out for fabrication. Samples were packaged in 40-pin DIPs and in the students' hands by mid-January 1979. As of the conference (January 22, 1979) three of the nineteen projects on the chip had been tested and found to work. The microprocessor described here will be tested in early February.

We intend to implement a full-scale version of a LISP processor in 1979, using essentially the same design strategies. The primary changes will be the introduction of a full garbage collector and an increase in the address space and number of types. We have tentatively chosen a 41-bit word, with 31 bits of address, 5 bits of type, 3 bits of "cdr code", and 2 bits for the garbage collector.

Conclusions

We have presented a general design for and a specific example of a new class of hardware processors. This model is "classical" in that it exhibits the stored-program, program-as-data idea, as well as the processor/memory dichotomy which leads to the so-called "von Neumann bottleneck" [Backus 1978]. It differs from the usual stored-program computer in organizing its memory differently, and in using an instruction set based on this memory organization. Where the usual computer treats memory as a linear vector and executes a linear instruction stream, the architecture we present treats memory as linked records, and executes a tree-shaped program by recursive expression evaluation.

The processor described here is not to be confused with the "LISP Machine" designed and built at MIT by Greenblatt and Knight [Greenblatt 1974] [Knight 1974] [LISP Machine 1977] [Weinreb 1978]. The current generation of LISP Machine is built of standard TTL logic, and its hardware is organized as a very general-purpose microprogrammed processor of the traditional kind. It has a powerful arithmetic-logic unit and a large writable control store. Almost none of the hardware is specifically designed to handle LISP code; it is the microcode which customizes it for LISP. Finally, the LISP Machine executes a compiled order code which is of the linearly-advancing-PC type; the instruction set deals with a powerful stack machine. Thus the LISP Machine may be thought of as a hybrid architecture that takes advantage of linear vector storage organization and stack organization as well as linked-list organization. In contrast, the class of processors we present here is organized purely around linked records, especially in that the instruction set is embedded in that organization. The LISP Machine is a well-engineered machine for general-purpose production use, and so uses a variety of storage-management techniques as appropriate. The processor

described here is instead intended as an illustration of the abstracted essence of a single technique, with as little additional context as possible.

We have designed and fabricated a prototype LISP-based processor. The actual hardware design and layout was done by Steele as a term project for a course on VLSI given at MIT by Lynn Conway in Fall 1978. The prototype processor has a small but complete expression evaluator, and an incomplete storage manager (everything but the garbage collector). A more complete description of the prototype processor, plus some exposition on the nature of LISP evaluators in this context, may be found in [Steele 1979]. We plan to design and fabricate by the end of 1979 a full-scale VLSI processor having a complete garbage collector, perhaps more built-in primitive operations, and a more complex storage representation (involving "CDR-coding" [Hansen 1969] [Greenblatt 1974]) for increased bit-efficiency and speed.

A final philosophical thought: it may be worth considering kinds of "stuff" other than vectors and linked records to use for representing data. For example, in LISP we generally organize the records only into trees rather than general graphs. Other storage organizations should also be explored. The crucial idea, however, is that the instruction set should then be fit into the new storage structure, in some natural and interesting way, thereby representing programs in terms of the data structures. Continuing the one example, we might look for an evaluation mechanism on general graphs rather than on trees, or on whatever other storage structure we choose. Finally, the instruction set, besides being represented in terms of the data structures, must include means for manipulating those structures. Just as the usual computer has ADD and AND; just as the LISP architecture presented here must supply CAR, CDR, and CONS; so a graph architecture must provide graph manipulation primitives, etc. Following this paradigm we may discover yet other interesting architectures and interpretation mechanisms.

Acknowledgements

We are very grateful to Lynn Conway for coming to MIT, teaching the techniques for NMOS design, and providing an opportunity for us to try our ideas as part of the course project chip. The text used for the course was written by Carver Mead and Lynn Conway. Additional material was written by Bob Hon and Carlo Sequin. It should be mentioned that the course enabled a large number of students to try interesting and imaginative LSI designs as part of the project chip. This paper describes only one project of the set, but many of these student projects may have useful application in the future.

Paul Penfield and Jon Allen made all this possible by organizing the LSI design project at MIT and arranging for Charles Botchek and Lynn Conway to teach.

Charles Botchek provided our first introduction to the subject and started our wheels spinning.

The course and project chip were executed with the cooperation, generosity, and active help of the Xerox Palo Alto Research Center, Micromask Inc., and Hewlett-Packard.

Dick Lyon and Alan Bell of Xerox PARC performed plots of the projects and assembled the projects into the final mask specifications. They were of particular direct aid to Steele in debugging his project.

Glen Miranker and William Henke maintained the plotting software used at MIT to produce intermediate plots of student projects during the design cycle, and were helpful in making modifications to the software to accommodate this project.

Peter Deutsch and Fernando Corbato were kind enough to hand-carry project plots from California to Boston to help meet the project deadline.

Tom Knight and Jack Holloway provided useful suggestions and sound engineering advice, as usual. (In particular, Knight helped Steele to design a smaller pad to reduce the area of the project, and Holloway suggested the probe multiplexor technique for testing internal signals.)

This work was conducted at the MIT Artificial Intelligence Laboratory. It was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

Guy Steele's graduate studies at MIT during 1978-1979 are supported by a Fannie and John Hertz Fellowship. In the spring of 1978 they were supported by a National Science Foundation Graduate Fellowship.

References

- [Backus 1978] Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Function Style and Its Algebra of Programs." Comm. ACM 21, 8 (August 1978),
- [Baker 1978] Baker, Henry B., Jr. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.
- [Berkeley 1964] Berkeley, Edmund C., and Bobrow, Daniel G. (Eds.) The Programming Language LISP: Its Operation and Applications. Information International, Inc. (Cambridge, 1964).
- [Conrad 1974] Conrad, William R. A compactifying garbage collector for ECL's non-homogeneous heap. Technical Report 2-74. Center for Research in Computing Technology, Harvard U. (Cambridge, February 1974).
- [Galley 1975] Galley, S.W. and Pfister, Greg. The MDL Language. Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).
- [Greenblatt 1974] Greenblatt, Richard. The LISP Machine. Artificial Intelligence Working Paper 79, MIT (Cambridge, November 1974).
- [Hansen 1969] Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation." Comm. ACM 12, 9 (September 1969), 499-507.
- [Hart 1964] Hart, Timothy P., and Evans, Thomas G. "Notes on implementing LISP for the M-460 computer." In Berkeley and Bobrow, The Programming Language LISP, 191-203.
- [Knight 1974] Knight, Tom. The CONS Microprocessor. Artificial Intelligence Working Paper 80, MIT (Cambridge, November 1974).
- [LISP Machine 1977] The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. LISP Machine Progress Report. AI Memo 444. MIT AI Lab (Cambridge, August 1977).

- [McCarthy 1962] McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).
- [Minsky 1963] Minsky, M. L. A LISP garbage collector using serial secondary storage. Artificial Intelligence Memo No. 58 (revised), MIT (Cambridge, December 1963).
- [Morris 1978] Morris, F. Lockwood. "A Time- and Space-Efficient Garbage Compaction Algorithm." Comm. ACM 21, 8 (August 1978), 662-665.
- [Saunders 1964] Saunders, Robert A. "The LISP system for the Q-32 computer." In Berkeley and Bobrow, The Programming Language LISP, 220-231.
- [Schorr 1967] Schorr, H., and Waite, W. M. "An efficient machine-independent procedure for garbage collection in various list structures." Comm. ACM 10, 8 (August 1967), 501-506.
- [Steele 1976a] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).
- [Steele 1976b] Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).
- [Steele 1977] Steele, Guy Lewis Jr. Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto. S.M. thesis. MIT (Cambridge, May 1977). operations.
- [Steele 1978a] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Revised Report on SCHEME. MIT AI Memo 452 (Cambridge, January 1978).
- [Steele 1978b] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two). MIT AI Memo 453 (Cambridge, January 1978).
- [Steele 1979] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. Design of LISP-based Processors. MIT AI Memo, forthcoming.
- [Sussman 1975] Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).
- [Wegbreit 1974] Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 23-74. Center for Research in Computing Technology, Harvard U. (Cambridge, December 1974).
- [Weinreb 1978] Weinreb, Daniel, and Moon, David. LISP Machine Manual (Preliminary Version). MIT AI Lab (Cambridge, November 1978).

COMPUTER-AIDED DESIGN SESSION

Chairperson:

William R. Heller, IBM Corporation, Visiting
Professor of Computer Science at Caltech.

This session comes in the middle of our five conference gatherings. In several respects, it also occupies a central position in the development of VLSI systems. First, new techniques and programs will be required. Second, the subject includes an increasingly essential and expensive set of tools for those companies designing, making and selling digital systems in the era of VLSI.

What are the new problems presented by VLSI to system designers and hence to design programming application and system developers? First of all, VLSI means that complex systems formerly assembled as boards, cards, chip carriers and chips are now becoming physically small enough to occupy only a few chips. Hence the former design procedure for even moderately large systems, which consisted in more or less delayed and loosely coupled iterations among a rather large group of system architects, gate level logic designers and package designers, has to be revised.

Second, the design tools can not be aimed at separated and simpler sub-problems as easily as before. When hundreds of thousands of circuit cells and signal wires share the same area with appropriate power distribution and pads, the design tools must be shaped so as to give an engineering balance among them at every stage of the design process.

Third, the ability to test the hardware during design (with emphasis on diagnostics) and after manufacture (with emphasis on low cost in

dollars and time) no longer is made easier by the physical separability of component parts. Testing strategy and tactics must be an ever more integral part of the design.

Fourth, the ability to make relatively rapid and precise changes and checks of the design is reduced in the hardware phase, so that it must be enhanced in the software phase.

Fifth and most important of all, the entire philosophy of system development and production must be such as to balance engineering design time against manufacturing bring-up and production times. If innovation and growth are to continue to characterize this computational age, the schedules and dollars for each system committed to production by industry must not grow out of bounds at the same time.

Our speakers will address their remarks to some of these problems.

Throughout we shall hear an emphasis upon the requirements and the techniques stipulated by the new VLSI design team - the system architect, the circuit and logic designer, and the chip layout expert. We shall not hear a well rounded discussion describing the parts of a finished design system suited for production of VLSI chips.

Rather we shall hear how some of the best industrial and university people formulate the problems, and suggest paths to solutions.

VLSI DESIGN METHODOLOGY

THE PROBLEM OF THE 80'S FOR MICROPROCESSOR DESIGN

Bill Lattin

Intel Corporation

Aloha, Oregon

ABSTRACT

The rapid evolution of semiconductor technology continues to make possible increasingly sophisticated electronic systems on single chips of silicon. By 1982, a single silicon chip is projected to have well over 100,000 transistors. This level of complexity represents a major problem for the VLSI designer in the 1980's. Unless there is a major change in design methodology, this level of VLSI technology will be grossly under-utilized due to the problems of design, layout and checking. With present design methods, a 100,000 transistor MOS chip will take 60 man years to layout and another 60 man years to debug.

INTRODUCTION

At present design rates, it is clear that the major problem for the 1980's will be to devise new design methodology in order that our rapidly evolving semiconductor technology, with all its density, will be widely usable by the electronics community. While technology has increased the on-chip complexity by a factor of four in the last two years, present designs are still based on methods that have not changed in the last six or seven years. This means, of course, that it now takes that much more of a manufacturer's resources to design each chip. In addition to the resources, the time to design, debug and transfer a complex microprocessor to production has increased at the same rate.

Clearly, the next challenge for manufacturers of VLSI devices will be how to reduce the resources and the time from conception to volume production of complex microprocessor chip families. While there are many aspects of this problem, which will require new methodology, this paper will focus on just one of these -- the portion of the design cycle known as "layout". This is the most expensive and time consuming portion of the design cycle and the one needing the most attention.

GROWTH IN COMPLEXITY

By using the number of active transistors on a chip as a general measure of complexity and by plotting it against the year of introduction of that microprocessor, one can begin to understand the magnitude of the problem.

Figure 1 shows the historical density improvement for microprocessor technology. Notice that the complexity of microprocessors at the chip level has grown exponentially for the last few years. This ever increasing number of devices on a chip will continue to make the layout portion of the design cycle the largest cost and time component. It could even be stated that the layout will become the dominant factor in development cost, and possibly, the limiting factor in chip design.

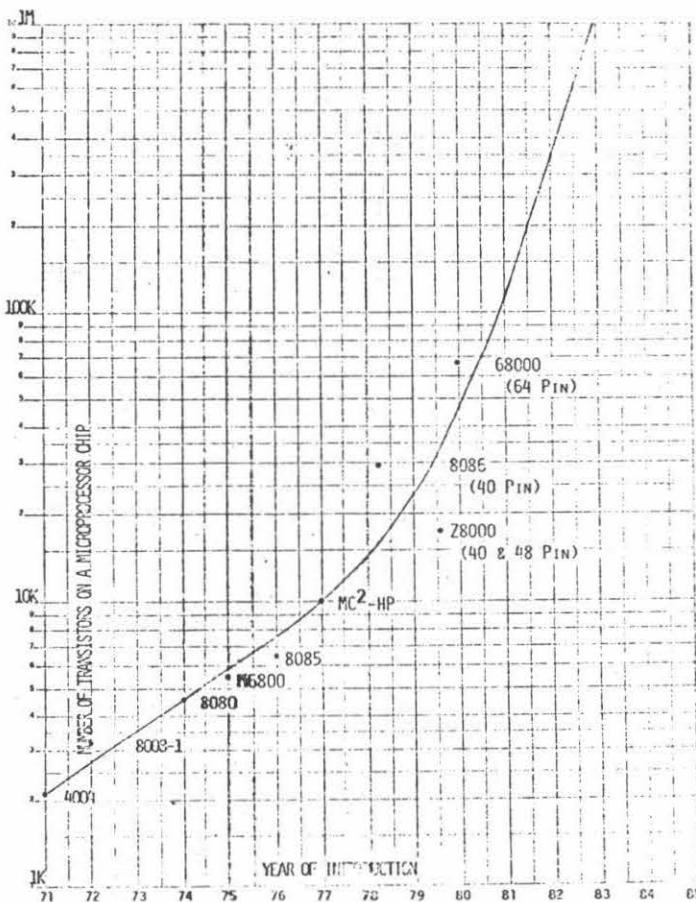


Figure 1

PRODUCTIVITY

Over the last few years, layout productivity has decreased as chip complexity has increased. This productivity, for a wide variety of layout techniques, including interactive drawing systems, is between five and ten devices per layout designer per day. This includes the time to draw, check and correct a layout.

If one assumes some productivity figure, a chart of estimated manpower can be derived from Figure 1. Figure 2 is such a chart and is derived from Figure 1 by using the number of transistors provided by the technology and translating that into man years of layout effort. The figure assumes that each layout designer can achieve the optimistic productivity level of ten transistors per day. The graph shows that a complex microprocessor in 1982 will take over 60 man years to layout. Since that level of layout effort would be almost impossible to manage, it means that the technology will have outrun the semiconductor manufacturer's ability to use it in a timely manner -- at least for complex logic chips.

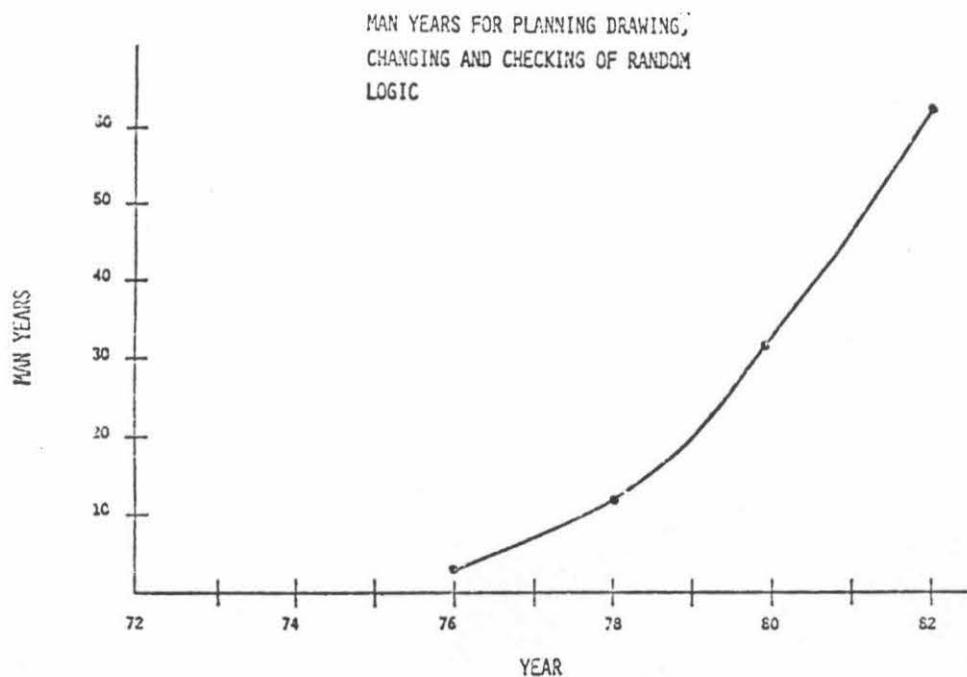


Figure 2

REGULARIZED STRUCTURES

It is important to note that Figure 2 does not apply to memory chips in that time frame since their layout is based on the design of a single cell which can be layed out once and then replicated many times automatically.

A similar concept has been proposed by Sutherland and Mead (Ref. 1) as a method to keep logic chip complexity from becoming unwieldy. It is a concept whereby random logic layout with its massive interconnection problems is structured by a set of well defined communication paths. These structures are usually implemented with very regular cells such as ROM's, RAM's and PLA's; although it is the geometric regularity of the interconnection between elements that provides the greatest benefit in reducing complexity. This paper would suggest a second major benefit of designing complex microprocessors with regular structures, and that is a decrease in layout time and effort. This decrease comes about because the use of regular structures reduces the total number of devices which must be individually drawn. In addition, the more structured layouts are easier to validate and check out.

To assess the impact of regular structures on layout time, we need a way to measure the degree of regularization on a given chip. The following parameter provides us with such a measure:

$$\text{Chip Regularization} = \frac{\text{Total Devices On A Chip}}{\text{Drawn Devices}}$$

where total devices includes all possible ROM and PLA placements, not just the bits actually programmed. The drawn devices are the devices that must be drawn, and therefore, require layout effort. This simple parameter matches an intuitive feel as to the degree of regularization as long as two rules are applied:

1. Data and program memories must be excluded when measuring the regularization of a single chip microcomputer.
2. The devices in all the chips must be included when measuring a multi-chip processor.

An indication of how regular past microprocessors have been is shown below:

$$8080 = \frac{4.6}{4.3} = 1.06 \quad 8085 = \frac{6.2}{2.0} = 3.10 \quad 8086 = \frac{29.0}{6.6} = 4.4$$

Analysis has shown that it should be possible to produce microprocessor designs with a regularization parameter between 10 and 20. If this can be achieved, it will reduce the layout effort in 1982 from 60 man years to 5 man years and allow us to utilize the technology to its fullest extent.

SUMMARY

The rapid evolution of microprocessor complexity, as well as the constant level of layout productivity, have caused manufacturers to reevaluate their VLSI design methodology. One part of this emerging methodology will be the use of regular structures in microprocessor design. In order to aid the designer in creating more geometric regularity into new chips, a measure of regularization has been proposed. This parameter would suggest that increasingly regular designs will not only solve the design complexity problem, but will also reduce the layout problem to a manageable level and thus allow microprocessor chips to fully exploit the new technology.

References:

1. Sutherland, I. and Mead, C.,
"Microelectronics and Computer Science",
SCIENTIFIC AMERICAN, September, 1977

Requirements for a Research-Oriented IC Design System

by

Jonathan Allen

Research Laboratory of Electronics
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Computer-aided design techniques for integrated circuits have grown in an incremental way, responding to various perceived needs, so that today there are a number of useful programs for logic generation, simulation at various levels, test preparation, artwork generation and analysis (including design rule checking), and interactive graphical editing. While the design of many circuits has benefitted from these programs, when industry wants to produce a high-volume part, the design and layout are done manually, followed by digitizing and perhaps some graphic editing before it is converted to pattern generation format, leading to the often heard statement that computer-aided design of integrated circuits doesn't work. If progress is to be made, it seems clear that the entire design process has to be thought through in basic terms, and much more attention must be paid to the way in which computational techniques can complement the designer's abilities. Currently, it is appropriate to try to characterize the design process in abstract terms, so that implementation and technological biases don't cloud the view of a desired system. In this paper, we briefly describe the conversion of algorithms to masks at a very general level, and then describe several projects at MIT which aim to provide contributions to an integrated design system. It is emphasized that no complete system design exists now at MIT, and that we believe that general design considerations must constantly be tested by building (and rebuilding) the various subcomponents, the structure of which is guided by our view of the overall design process.

We see these overall design processes as a set of conversions through a succession of representations, starting at the level of an algorithmic description and proceeding to a mask specification via a set of transformations. Each of these representations is keyed to a

particular design focus. Thus, for example, there is an initial level at which the algorithm must be represented, and following levels where the architectural structure, detailed logic design, electrical circuit, and geometrical layout must be described. There is no unique partitioning into levels, so that some systems may want to dispense with an architectural description, while others may want to add a topological layout level. Clearly, these levels pose vastly different entities that must be characterized, so that each of the levels requires a design language appropriate to the semantic primitives (e.g. logic gate or circuit node) that are relevant at that level. Despite this diversity of representation, however, the transformations that are used to convert between them must preserve the original algorithmic intent. That is, the constraints that hold in the original algorithmic description (which must, of course, be represented in some appropriate linguistic form) must also hold at all succeeding levels, although they will certainly have to be interpreted in terms of the entities native to that level. Thus, the final mask specification must be a realization of a scheme to execute the original algorithm, and while the algorithm is expressed by some linguistic means with its own set of relevant structures to be manipulated, the mask specification will have to represent the base for task execution in terms of static geometrical structures. It is clear that a major part of the task of building a design system is the determination of the levels at which representations are desired, followed by a clear semantic definition of what the semantic entities are at that level and the (limited) ways in which they can be manipulated. Transformations across levels must then express how these semantic entities link up. We feel that it is important to conceive of the design process in terms of such linked programs with clear semantics, and that representational issues (such as common data bases) must follow from the more basic linguistic specifications.

From the start, we have felt that the biggest weakness in the present arsenal of IC design tools has been the lack of appropriate linguistic means to specify algorithms. Certainly algorithms are frequently described in a common programming language, such as ALGOL, but for purposes of IC design, such representations are inappropriate. Programming languages have generally been devised to specify algorithms that will be executed on a single-sequence (e.g. von Neumann) computer, and the usual kinds of statements referring to the time or space requirements of an algorithm assume such an architecture. In hardware systems, however, there is much opportunity for parallelism, and so the usual procedural representation of an algorithm is inappropriate. What is needed, then, is some means to represent the underlying "competence" of the algorithm, devoid of any performance bias, so that only those constraints that are common to all algorithms that execute the given task contribute to this "deep structure". This permits us to then explore the ways in which the task can be executed as a separate problem, and hence provide the designer with a flexible means to pick the implementation that gives

the best space/time tradeoff for the particular application at hand. We have mounted two attacks on this problem. In one, the job has been to find the underlying structure common to all equivalent algorithms, while in the other, we have looked for interactive means to explore the various algorithmic alternatives. These two projects are thus complementary, and we discuss them next.

Building from considerable experience with the construction of computer-based debugging tools for electrical circuit analysis, it has become clear that the construction of constraint diagrams is very useful for encapsulating what is known about a circuit so that inferences can be made about related parameters in a circuit. Typical constraints include Kirchoff's voltage and current laws and Ohm's law, the latter expressing a constraint between the voltage across and the current through, a resistor. Constraint diagrams for electrical circuits can be readily drawn, so that inferences can be made about circuit variables which control an observed value. One way to think of the constraint diagram is to regard it as an expression of the enduring, time-independent truth about the circuit. That is, the constraints characterize what must hold in the circuit under any parametric conditions.

Next, we observe that this notion of constraint should be useful for any system that performs a task, and since the constraint representation characterizes what must hold during any execution of the given task, it provides a common "deep structure" for all possible ways to perform the task. The neutral constraint representation thus provides an ideal starting point to begin the design of an integrated circuit. The designer is free to pick out any implementation consistent with the algorithmic constraints that will provide the required performance.

Attractive as this view may be, there are at present several missing pieces limiting the realization of its potential. We need to establish that constraint networks can be created for a broad and significant class of algorithms, and that an appropriate formalism emerges for this purpose. We also need to develop a variety of techniques for building performance strategies on the constraint representations, and hence provide a means to project several space/time performance means from a single underlying constraint network.

Constraints are of course a natural way to represent design rules, and we are working toward the structure of a broader class of rules than just those specified in terms of the layout geometry. Constraints such as current density, power, speed, capacitance, and other performance parameters should be integrated with those expressed at the layout level. This is an example of a general formalism being used to control the design at several levels of representation, a capability which is likely to be increasingly useful.

Constraint network representations emphasize the multiplicity of ways in which a task can be performed, so it is natural to ask how the designer can select a particular performance strategy. One way to do this is to start with some algorithm for performing the task, and then provide means for changing its performance via a set of transformations on the original algorithm expression. We have constructed linguistic techniques for originally specifying an algorithm, and then transforming it into a number of forms which vary throughout the space/time design space. For example, some algorithm blocks consist of statements that can be executed in any order, and hence can be done in parallel (p-blocks), while still other blocks contain statements, all of whose right-hand-sides should first be evaluated, and then all assignments should be done in parallel (c-blocks). Transformations have been defined to convert between any pair of these blocks, thus allowing the designer to explore various algorithms which are equivalent in terms of input/output behavior. Standard do-loop and for-all control structures will also be provided. Once the desired architecture is obtained, then it should be compiled into logic, represented by a low-level hardware design language.

Space and time are the fundamental architectural factors which can be traded. We expect that the requirements of IC design systems will cause the nature and control of these tradeoffs to be heavily studied. It is interesting that the design freedom presented to the IC designer makes these considerations much more prominent for large circuits than they have been in the past. At the current stage of research, we feel that it is appropriate to give the designer full control of these tradeoffs. Possibly in the future the nature of design may be sufficiently understood to encapsulate the design exploration process in a program, but we are not there yet.

From the architectural level, representation at the logic and circuit level must be obtained, leading to the layout and artwork level. We have not focused our attention on the logic and circuit levels, but we have felt that the geometrical level provides problems in the management of complexity that need immediate attention. Large integrated circuits must take advantage of modular subcomponents, or macrocells, and these have to be integrated into the semantic representations at the various levels. In recent years, the initial standard cell approach, using small pre-compiled functions arrayed in rows, has yielded to more general and larger cells, representing memories, ALU's, and other structures, which enjoy complete freedom of placement on the chip surface. This escape from layout rigidity has led to the design problem of specifying placement for these cells, and providing the required interconnect. A particular problem of interest is the extent to which hierarchical grouping of these cells should be automatically enforced as opposed to allowing normal grouping. We feel that it is important to benefit from previous designs, and to permit fast design of large systems based on these modules, even if

substantial inefficiencies remain. Perhaps, by delaying the binding of the geometrical layout of these cells, topological representation can be stored for them in the designer's library, providing flexibility of connection and shape which can be made responsive to the overall layout constraints of the circuit. Of course, the ways in which these topological specifications are bound into geometry affects the corresponding circuit performance, so that as the geometry is instantiated, it is necessary to enforce the kind of extended design rules discussed above which not only restrict placement on the surface, but also constrain the circuit performance. Rapid artwork analyzers are needed to keep the geometrical and circuit representation linked up and related to the design constraints. The circuit representations must permit ready analysis and simulation for a detailed view of performance. Thus we feel it is extremely important to view the layout problem in terms of all its consequences, and to provide high-speed interactive means to explore these relationships. So just as we have placed heavy emphasis on the initial pole of the design trajectory, i.e. the algorithmic specification, we are also focusing heavily on the set of issues tied to the binding of layout geometry at the other final pole of the design process. Both areas require flexible, semantically clean representations, with options for binding of decisions available interactively to the designer.

The need for highly interactive computing facilities, particularly for graphic editing of the various representations, has led us to build an extended version of the MIT AI Laboratory LISP computer. This machine is designed as a personal computer, capable of providing very fast response. It has a large (24-bit) address space, and gives performance competitive with many large time-sharing systems. Two graphic bit map displays are provided, one for black and white, and one for color. These displays are refreshed from main memory while the computer is running, so that the displays are updated immediately as new data is computed. Since the two displays run simultaneously, large figures can be displayed on the black and white monitor while detailed views are available on the color display. The computer provides 128K 32-bit words of main memory plus 16K words of writable control store and a 300 megabyte disk. Both keyboard and "mouse" are supplied for manual interaction. This machine is connected by a local network to several other machines at MIT, so that large data bases and control services such as printing can be accessed. LISP is seen as an excellent language for the construction of data structures as well as interpreters at a variety of levels, and we feel that the flexibility provided by the combined hardware and software resources is important during research on the structure of an IC design system. A DECSYSTEM-20 computer is also available, and several simulation, artwork analysis, and plotting programs are used on this facility, which is also tied to the local network. In time, we expect that several LISP machines may be devoted to the IC design effort, all linked together, and to other large machines for data base

management and less interactive tasks using established programs available within the research community. Our view is that computing facilities of this type are essential to the design of a comprehensive design system, and will form the base of practical systems of the future.

The emphasis which we have placed on linguistic specification and manipulation of representations coupled with layout editing and highly interactive graphical computing must lead, of course, to a complete design system which permits the designer to convert an algorithm to a set of mask specifications. In order to focus these efforts, we are designing several circuits which should provide insight into the design system requirements. These include a speech synthesizer, a key encryption system, and a local network interface. Each of these projects is based on an original MSI design of between 120 and 150 dips, and each algorithm designer is also working on aspects of the IC design problem. We are even designing a microprocessor to serve in a next-generation LISP machine, so that our IC design efforts are reinforcing the tools available for such work. The study of various aspects of a design system, such as circuit simulation, may also lead to the design of special computing facilities for these tasks, particularly when these processes are time-consuming and expensive on conventional machines.

Acknowledgement. The IC design research described in this brief overview is part of a project devoted to means to convert algorithms to integrated circuits sponsored by AFOSR, of which the author is the principal investigator. Gerald Sussman and Guy Steele have been studying constraint systems, and the work on algorithm transformations is contained in Glen Miranker's doctoral thesis. Ronald Rivest and Andrea LaPaugh are working on generalized placement and routing, and Paul Penfield is devising artwork analysis algorithms. Clifford Fonstad and Dimitri Antoniadis have been interested in new device technology and process modeling, while the LISP machine has been designed and built by Tom Knight, Jack Holloway, and Richard Greenblatt. The cross-disciplinary interaction of these people has led to a number of refreshing insights, and the start of many new research interests motivated by the needs of IC design.

Hierarchical Design for VLSI: Problems and Advantages.

by

W.M. vanCleempot
Computer Systems Laboratory
Stanford University
Stanford, California 94305.

ABSTRACT

This paper describes the hierarchical design process for VLSI circuits and discusses the potential benefits and disadvantages.

1. INTRODUCTION

Over the past decade software designers have learned to cope with increasingly complex programs. In order to deal with this complexity a structured programming methodology has been developed. The advent of VLSI technology has brought similar complexity within the reach of hardware designers.

The objective of this paper is to explore the hardware design process and the problems that hierarchical design approaches create as well as their advantages.

Figure 1 shows the major steps and interactions in the hardware design process. The designer starts with a set of initial specifications that are often incomplete and possibly incorrect. The designer makes a number of design decisions both in terms of logic design and physical design. In the mean time he also modifies or refines the specifications until a final design is reached.

Design decisions by the designer can take one of two major forms:

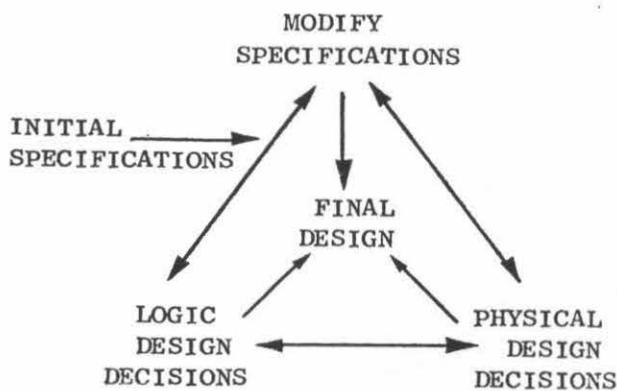


Figure 1: The Hardware Design Process

1. top-down decomposition of a behavior specification into less complex behavior specification modules.
2. bottom-up combination of physical building blocks into larger building blocks.

At some point in time the designer has to map the behavior specifications into some of the building blocks and to assure himself that this mapping will indeed result in a correctly operating design.

From the previous discussion it is clear that the design process is a combination of top-down and bottom-up processes that are happening concurrently in the designer's mind until he reaches a final correct design.

In an idealized model of the design process, there are three concurrent tasks:

1. behavior design, where the designer decomposes the initial specification into subproblems, possibly refining the specification by doing so.
2. structural design, where the designer tries to realize a block or module by the interconnection of more primitive modules.

3. physical design, where the designer tries to realize his design in a given technology.

In most cases the behavior and structural design processes go on concurrently, while the physical design process is done separately. It is, however, quite clear that the physical design process can have a tremendous influence on the behavioral or structural design of a system, necessitating many iterations during a design.

The physical design process itself may also be hierarchical in nature: a system can be partitioned into physical subsystems that in turn can be partitioned into physical subsubsystems and so on.

From the previous discussion on the digital system design process, one can conclude that there exist at least three major design hierarchies: a behavioral hierarchy, a structural hierarchy, and a physical hierarchy. The mapping of a behavior into a structural hierarchy is usually known as the logic design process, while the mapping of a structure into physical hierarchy is usually known as the physical process.

2. MAJOR DESIGN HIERARCHIES

2.1 BEHAVIORAL HIERARCHY

Figure 2 shows a possible hierarchical decomposition of a general purpose computer. In this particular example the CPU process consists of a fetch cycle and an execute cycle process. The fetch cycle process consists of instruction fetch address calculation and operand fetch subprocesses, while the execute cycle process consists of arithmetic and logical subprocesses. The arithmetic process consists of addition and subtraction. Each of those may consist of integer and floating point operations.

This is clearly an example in which the specifications did not pay attention to the possible physical implications of the hierarchical decomposition.

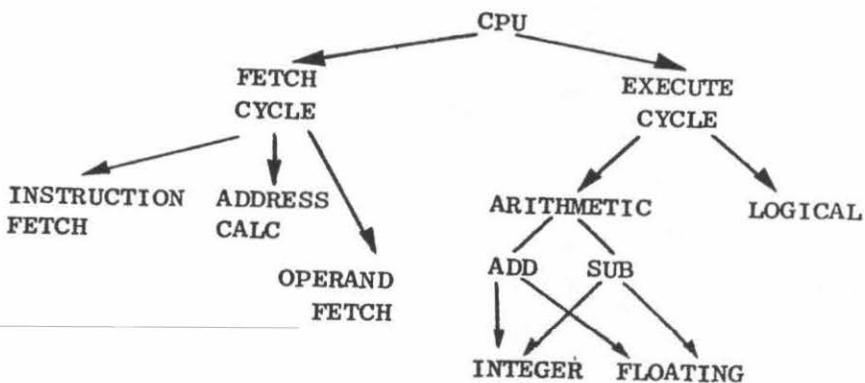


Figure 2: Behavior Hierarchy

A behavioral hierarchical decomposition can be implementation independent. For instance, an ISPS-like description [Ba77] of the IBM/370 architecture would be the same for all IBM/370 implementations.

One system that has adapted this hierarchical decomposition of a behavioral description is the SARA system [Es78] at UCLA. The behavior of a system or subsystem at every level of the hierarchy is modeled in terms of GMB graphs. The hierarchy of behavioral descriptions of a hardware design is similar to structured programming techniques.

This hierarchical behavior design process provides for a iterative refinement of the initial design specification.

Design automation tools that could be used during this phase of the hardware design process are: formal verification of the behavior specification at various levels of the hierarchy, as well as simulation of the design at various levels.

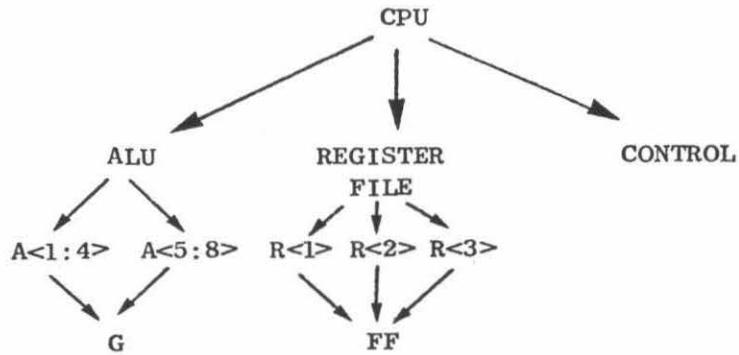
2.2 STRUCTURAL HIERARCHY

Figure 3: Structural Hierarchy

A possible structural hierarchy for a general purpose computer system is shown in Figure 3, where the CPU consists of an ALU register file and a control section. These three subsystems are interconnected by data paths which are not shown explicitly in this hierarchy. The ALU may consist of two identical sections, one for the first four bits and the second one for the last four bits. Both sections of the ALU are made out of gates of type G. The register file on the other hand consists of three registers R₁, R₂, R₃, which are constructed out of flipflops of type FF.

The SARA system at UCLA [Ga74] also includes a structural modelling tool, called SLL (a structural description language). This language allows the designer to specify the structural hierarchy of the design.

In the SCALD design system [MW78] a similar methodology is employed. The SCALD system was used for the design of an actual machine, the Stanford-1 processor. This processor

was designed within a few man-years (a fraction of the time normally spent on such a design project). One of the advantages of structural hierarchical design is that one has a structural modularization of the design with well-defined interfaces. This can reduce the design time considerably. The structural hierarchy of a system is developed concurrently with the behavioral hierarchy. At every level of the structural hierarchy one can associate a behavior description with every module. This behavior description itself can be hierarchical in nature, as was discussed in the previous section. This multitude of behavior hierarchies can be used for formal verification of the design decisions the designer makes.

Another function that one wants to perform during the structural design is to verify the intended behavior of a design against the structure that one is proposing. This so-called dataflow verification was implemented in the LCD system at IBM [OE77].

A final important aspect of a structural hierarchy is the fact that one can do a hardware macroexpansion of a design into lower level primitives. This provides an alternative to a direct hardware compiler from a behavior description into physical hardware. Since such a mapping is totally user-controlled, one can achieve very satisfactory results, as was demonstrated in the SCALD system [MW78].

2.3 PHYSICAL HIERARCHY

In order to package a system the designer has at his disposal a hierarchy of cabinets, racks and printed circuit boards. In integrated circuit design he may have at his disposal an hierarchy of supercells, macrocells, simple cells and transistors (Figure 4).

When designing large scale integrated circuits the human designer has a natural approach to lay out a design first in global terms, and then to successively refine this design down to the transistor level.

In order to cope with the complexity of VLSI circuits, several hierarchical IC design systems have been described

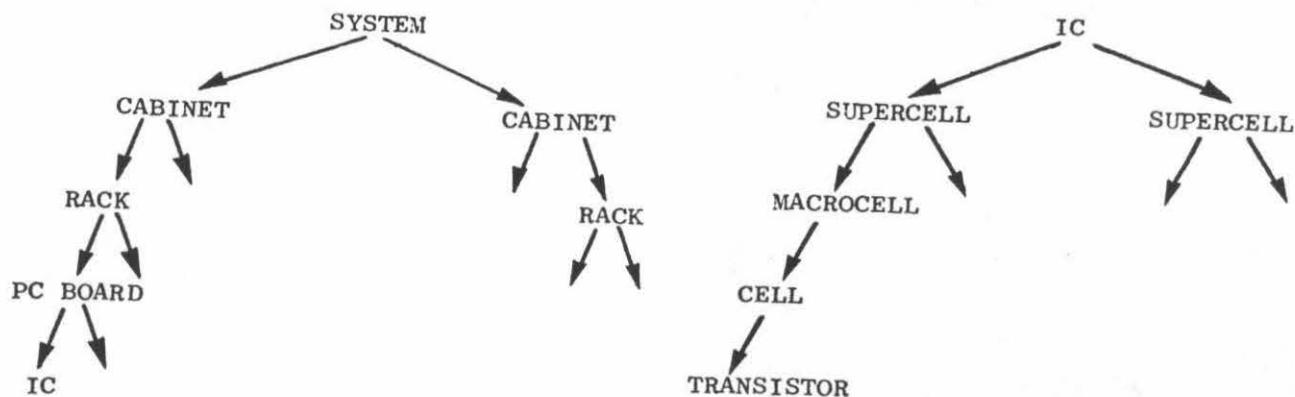


Figure 4: Physical Hierarchy

[PG78, vS77]. The potential advantage of these systems comes from reducing the amount of design time, while maintaining a reasonably efficient area utilization.

During the physical design process extensive use is made of design automation tools. A subtask of mapping the structural design hierarchy into a physical hierarchy is known as the partitioning process. This partitioning is usually done by human designers and very few automatic algorithms are used.

During the physical design process one can obtain sufficiently detailed information to allow a detailed behavior prediction of a system. This predicted behavior can then be verified against the intended behavior as it was specified by the designer during the behavior specification process.

3. ADVANTAGES AND PROBLEMS

3.1 DEMONSTRATED ADVANTAGES

A major advantage of hierarchical design methods is the reduction of design time. This technique is often used informally by IC layout designers: a cell is designed once and then replicated several times; further, IC designers often use a top-down planning phase to determine the global layout of a circuit. However, today's use of hierarchical layout by IC designers is totally informal.

Formal hierarchical methods for IC layout were proposed in [vS77] and [PG78]. Preliminary results show that hierarchical automatic layout yields results that are superior to single-level (classical) automatic IC layout algorithms [Pv79a, Pv79b].

The most convincing evidence that hierarchical design can reduce design time can be found in the use of the SCALD system for the design of the Stanford-1 processor [MW78]. The SCALD system uses a hierarchy of structural (connectivity) diagrams, specified graphically by means of SUDS (the Stanford University Drawing System). The SCALD system further consists of a macroexpander, which produces a wirelist at the physical module level and of a physical design system, which automatically produces the wirewrapped design. The amount of time and the number of engineering changes to the Stanford-1 design were at least an order of magnitude smaller than those of comparable ECL-based machines.

3.2 POTENTIAL ADVANTAGES

If one were to formalize hierarchical design one could require the designer to specify intended behavior for every module in the structural hierarchy.

The first advantage of such an approach would be extensive documentation of the design, now a major cause for misunderstanding and hence design errors and iterations.

If one were given the intended behavior and the structure (realization) of a system at a given level as well as the intended behavior of all subsystems at the next lower level, then formal verification could be used to check the consistency of the designer's decisions.

3.3 HIERARCHICAL DESIGN PROBLEMS

3.3.1 The Difference in Hierarchies

One of the important problems that one has to solve in hierarchical design is the mapping from a behavioral hierarchy into a structural hierarchy, and from a structural hierarchy into a physical hierarchy. In the past this mapping has always been done by human designers with little or no assistance from design automation tools. Due to the increased complexity, the need for formal verification tools as well as synthesis tools becomes more and more apparent.

One possible solution is to avoid the mapping problem by making every behavioral module the same as every structural module and the same as every physical module. This may seem a good solution at first sight, but it has some farreaching implications. For instance, a physical design decision may be made that could require changing the behavior specification or the structure specification of a design. Such a change may invalidate all the verification and simulation results that were obtained on the behavior or structural design. This may be a non-acceptable solution that may result in a large number of iterations during the design process. If the hierarchical structure of the behavior of a design is not the same as the structural or physical hierarchy then automated or computer-aided mapping processes are needed. The main objective of these automated mapping tools would be to speed up the time required to perform an error-free mapping of behavior into structure. Unfortunately, very little work has been done on this problem and it is not clear today how feasible such an approach would be.

3.3.2 Circuit Layout Problems

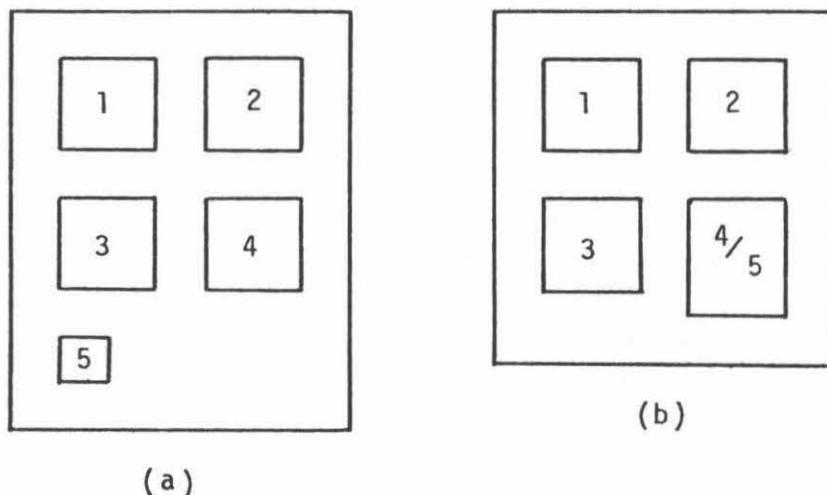


Figure 5: Inefficient Layout due to Hierarchical Process

Figure 5 illustrates the problem of mapping a structural hierarchy into a physical hierarchy. In this example, which represents a simple integrated circuit layout, the design consists of four cells of equal size, numbered one to four, and a fifth cell which is considerably smaller. If one were to consider a one-to-one mapping from the structural hierarchy into a physical hierarchy, the layout of Figure 5a would result with as a consequence a potential inefficient area utilization. However, if in the physical design process one could combine 4 and 5 into a physical module, then the physical design may be much more compact as shown in Figure 5b. If behavior, structure and physical hierarchies were identical then this design decision made during the physical layout of a circuit would have repercussions on both the structural and behavior specifications which may have been verified. Nonetheless the decision made here should have no effect on either the structure or the behavior of the circuit since it is a purely physical design decision.

In a purely hierarchical approach one encounters the problem of optimal shape determination for the blocks in the

layout hierarchy. For example, in the structural hierarchy of Figure 3 all registers are built out of flipflops of type FF, that are physically identical. To associate a single physical design with a structural module can be very inefficient.

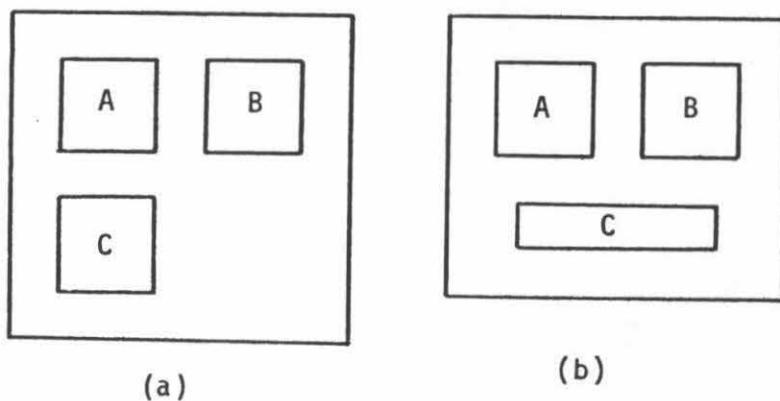


Figure 6: Influence of Different Physical Layout

Consider the example of Figure 6, where the layout consists of three structurally identical modules A, B and C of type FF. The layout could be greatly improved if the physical shape of C were changed as in Figure 6b.

A similar problem of efficient area utilization exists with the assignment of terminals around the periphery of a module. Figure 7a shows two identical modules A and B, with the specified connections. If the order of the terminals could be modified as in Figure 7b, a more efficient layout could be obtained. This however would necessitate the redesign of all modules, thereby greatly reducing the potential benefits.

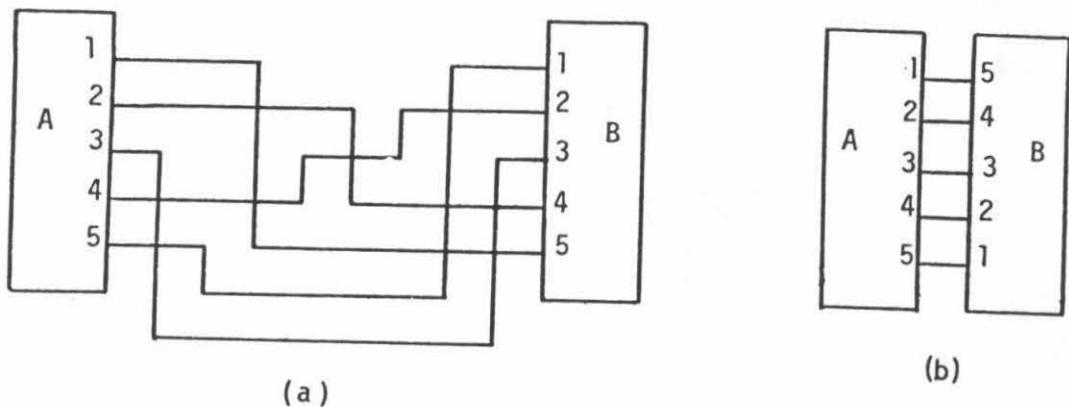


Figure 7: Influence of Terminal Assignment

3.3.3 Multiplicity of Hierarchical Representations

It was already pointed out that a large number of hierarchical behavior descriptions may exist, complicating the problem of formal design verification.

In a similar fashion, several structural hierarchies may exist, as pointed out in [va77]: for the purpose of simulation using a zero-delay, three-valued simulator, a flipflop may be modelled by a collection of gates and delay elements; for use with another simulator, this same module may be mapped into NAND gates with variable delays and for the purpose of IC layout the mapping would be into a set of CMOS or NMOS transistors. In other words the structural hierarchy of a design depends not only on the designer's decisions relating to price-performance trade-offs, but also on the actual purpose of the description.

In a similar fashion, the physical hierarchy of a design is not unique, but rather a choice of the designer.

The problem of choosing the best hierarchy cannot be solved by the hierarchical design process concept. It is important to realize that hierarchical design can merely produce an acceptable design in a reduced amount of time.

3.3.4 The Need to Exceed the Boundaries of a Level

In the previous section, several examples were given that illustrated the need for look-ahead over one or more levels of the hierarchy.

Most of today's design automation software works on a fixed level of abstraction. This is partly due to the lack of hierarchical concepts in these tools and partly due to the nature of the problem they are trying to solve. Among the latter we mention the layout of a PC board, the calculation of timing delays in a LSI circuit, the calculation of physical wirelength of a design.

For these examples, a simple macroexpansion of the design provides a solution. The basic idea of a macroexpansion is to collapse several levels of a hierarchy into a single level of abstraction.

However, there are situations in which interactions between the levels of a hierarchy are more complex and hence tend to counteract the advantages. One such case is the problem of geometrical design rule adherence in LSI layout. These design rules are usually expressed as a combination of complex relationships on rectangular or polygonal elements. In a true hierarchical design environment, one has to define design rules between cells that are far more conservative, hence resulting in a less optimal layout.

A similar problem exists with IC layout: a cell is often seen as a closed polygon without allowing use of the inside area for laying out a collection of cells. In reality some of this intra-cell area could be used for optimal layout.

4. CONCLUSIONS

In this paper we have attempted to present a model for the human designer and the design process. We have postulated that a hierarchy of behavioral specification, structural implementation and physical realization is a reasonable model for the human designer.

Design automation tools should capture this hierarchical information from the human designer and use it for making more intelligent design decisions. An open problem in design automation is the problem of mapping from a behavioral specification hierarchy into a physical implementation hierarchy, and from a physical implementation hierarchy into a physical realization hierarchy. This process should be either automated or computer-aided, if we want to deal with complex systems.

Hierarchical design is capable of reducing the amount of resources devoted to a design. As always this yields a trade-off between design optimality and design time. In the VLSI era, a suboptimal inexpensive design may be more important than a more expensive but more optimal design.

In some cases there will be a need to perform a macro-expansion on the design in order to consider the whole design at a single level.

REFERENCES

- [Ba77] Barbacci, M.R. "The ISPS Language," Carnegie Mellon Univ., Dept. of Computer Science, Technical Report, 1977.
- [Es78] Estrin, G. "A Methodology for the Design of Digital Systems , supported by SARA at the Age of one," Proc. National Computer Conf., Anaheim, Cal., June 1978, pp. 313-324.
- [Ga74] Gardner,R. "A Methodology for Digital System Design based on Structural and Functional Modelling," Ph.D. Thesis, UCLA, 1974.
- [MW78] McWilliams, T. M., and L. C. Widdoes, Jr., "SCALD: Structured Computer-Aided Logic Design," Proc. of the 15th Design Automation Conference, Las Vegas, Nevada, June 1978, pp. 271-277.
- [OE77] Ofek,H.; Evangelisti,C.J. and Goertzel,G. "Designing with LCD: Language for Computer Design," Proc. 14th Design Automation Conf., San Francisco, June 1977, pp. 369-376.
- [PG78] Preas,B.T. and Gwyn,C.W. "Methods for Hierarchical Automatic Layout of Custom LSI Circuit Masks," Proc. 15th Design Automation Conf., Las Vegas, Nevada. .June 1978, pp. 206-212.
- [Pv79a] Preas, B.T. and vanCleemput,W.M. "Placement Algorithms for Arbitrarily Shaped Blocks," Proc. Int. Symposium on Circuits and Systems, June 1979, to be published.
- [Pv79b] Preas, B.T. and vanCleemput,W.M. "Routing Algorithms for Hierarchical IC layout," submitted for publication, 1979 Design Automation Conf.

[vS77] vanCleemput,W.M. and Slutz,E. "Initial Design Considerations for a Hierarchical IC Design System," Conf. Record 11th Asilomar Conf. on Circuits, Systems and Computers, November 1977, pp. 334-341.

[va77] vanCleemput,W.M. "An Hierarchical Language for the Structural Description of Digital Systems," Proc. 14th Design Automation Conference, New Orleans, Louisiana, June 1977, p. 377-385.

DATA BASE CONSIDERATIONS FOR VLSI

L.W. Leyking
California Institute of Technology
and Burroughs Corporation

ABSTRACT

This paper will discuss the motivations and history of data base design for design automation. The objectives of using a data base for VLSI design are outlined and the aspects of merging the logical and physical data for a VLSI design are proposed. A hierarchical data base structure and model is presented and typical data examples of logical and physical VLSI data are mapped into this structure.

1.0 Introduction

The issue of data base design for VLSI is of paramount importance. The success of how well designers can model, describe and capture the logical and physical data for a complex chip design will, in part, be measured by the cost and speed of producing new designs in the market place. Companies doing VLSI design will place more emphasis on how fast a product can be brought into the market place to produce a competitive advantage. A well structured design data base will help in achieving this goal.

2.0 Data Base Systems and Definitions

Considerable effort was applied to the data base problem in the mid 60's by the Conference on Data Systems Languages (CODASYL). Work began in 1965 to define the problems and a draft of the work was completed in 1969. Some of the early data base systems were plagued with efficiency problems and lack of capability and it wasn't until the early 70's that most computer manufacturers were offering reliable DBMS as part of their system software. Table 1 charts some of the more common DBMS as of function of the time they were introduced. [1]

Most early DBMS were organized around either a sequential or tree (hierarchical) structure but more recently both network and relational data bases have been proposed and implemented. Figure 1 indicates the four basic types of data structures. Sequential structures consist of records linked together by forward and backward pointers to facilitate tracing from one record to

TABLE # 1
DATA BASE MANAGEMENT SYSTEMS
HISTORY

TIME → (YEAR)															
65	65	67	68	69	70	71	72	73	74	75	76	77	78	79	80
BEGIN CODASYL WORK				CODASYL DRAFT		DMS-1100		DMS/90 DBMS/10		DMS 170 IDS 11		DBMS-11 SIMDBM		RELATIONAL DBMS	
IMS	TOTAL				SYSTEM 2000			IBMS SIBAS DMSII							

FIGURE 1
DATA BASE
RELATIONAL

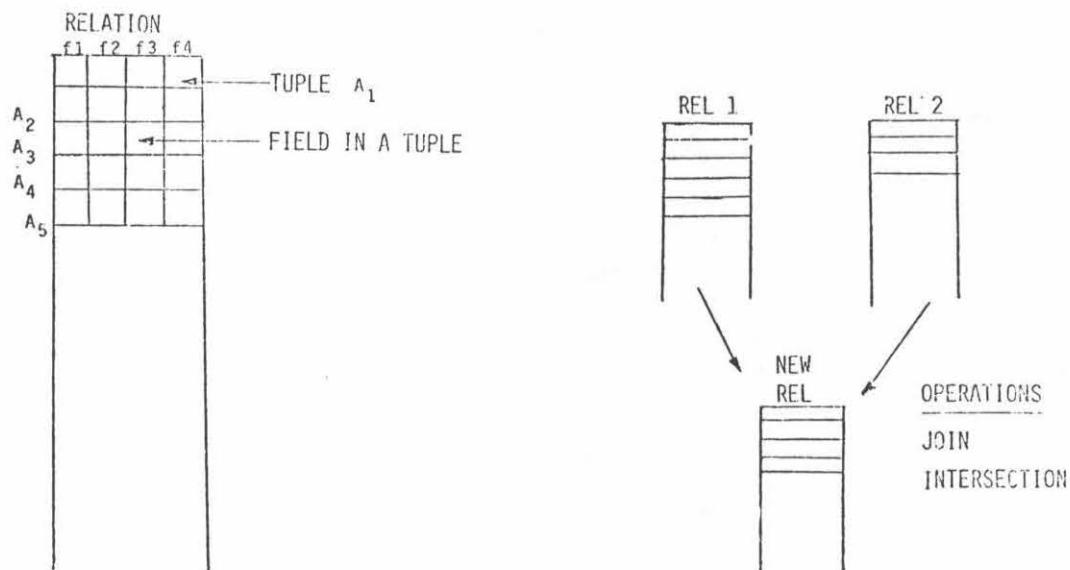
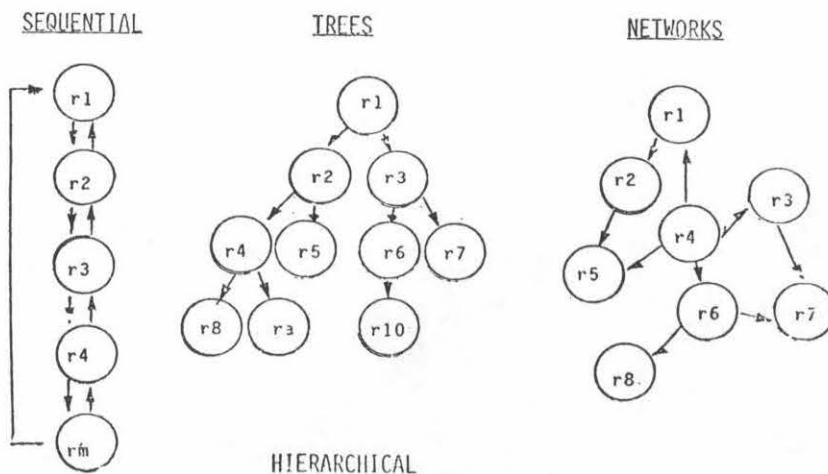


FIGURE 1
DATA BASE
STRUCTURES



another. These records were often sorted on a basic key item which allowed rapid lookup techniques such as binary search. Hashing schemes were often employed to store actual records on magnetic disk in specific pages or segments. Tree structures were used when data organizations dictated a node, branch, and leaf hierarchy with a top down mode of access. Network structures allowed access from any node to any other node in the network and gave the advantages of a tree and sequential structure. Recent data base developments have seen the introduction of relational structures. Individual relations can have basic operations such as join, union, intersection, and projection performed on them to yield new relations. The new relations can be organized to facilitate a particular program's operation. Relational data bases offer flexibility at the cost of efficiency in organizing the data.

Each type of data base structure must be analyzed against the task to be performed. No one structure is best for all problems although in VLSI design automation tasks, the tree and relational structures appear promising.

3.0 Objectives and Motivations of a VLSI Data Base

The need for a VLSI data base is shown by the increased importance VLSI circuit design is taking in the engineering community. A design revolution is underway where more digital computer system functions are being incorporated into VLSI chips, with the end goal of having complete systems on one chip. We are already at this stage with the introduction of microprocessors on a chip and the future indicates even more single chip systems which incorporate more memory and control logic. Today's system chips range in size between the 10K to 30K device (transistor) category with indications of chips approaching 100K devices in a few years. To be able to work with this number of devices, designers will need new data bases to aid in the organization and hierarchical top down structuring of VLSI chip designs.

The following paragraphs outline several major objectives of a VLSI data base, some of which will be essential even in an experimental design system.

1. Hierarchical Structure

To be effective in reducing the amount of design data to a minimum, the design must be hierarchically broken down into smaller and smaller cells which are complete unto themselves. The word complete in this sense means complete in terms of containing all logical and physical data which represent the cell. No additional data should be needed to carry out the VLSI DA design process in terms of simulation, partitioning, interconnection, design rule checking or mask fabrication.

2. Manipulate Large Chips

The data base should be capable of manipulating one or more chips with device numbers approaching 1 million. It is interesting to note how many levels in a hierarchical design are needed to achieve a 1 million device chip. For discussion purposes I will choose a hypothetical design with 5 levels as depicted in Table 2. Level 1 is the highest level cell, the system chip, and level 5 is the lowest level, the primitive cells. Looking at the number of typical subcells/level and the typical number of devices/cell we see that chips between 100K and 2.5M devices can easily be achieved in 5 levels of hierarchy. This number of levels is consistent with the number of levels used in industry to achieve the present day chips. The data base should not be limited in levels, but we do not expect to see 20 levels in the hierarchy, more typically 4-10.

3. Simple to Use

The data base should be simple to use both by the user and the applications programmer. Two sub-objectives could be considered.

A. Using the Data Base Directly

This objective could be desirable as each function program would then not have to build a run time data base by extracting data from the data base but could use the data base directly. Whether this is practical will be determined by the requirements of each function and how it maps to the data base structure. Minimum data redundancy would be achieved using this method.

B. Using the Data Base as a Repository for Data Only

This objective would mean each application program would extract its data from the data base at run time, build its own internal data base for run time efficiency and then deposit any new data back into the data base at run completion. This method would create data redundancy by the application program but would achieve greater run-time efficiency due to the building of special run-time data structures.

4. Non-redundant Data Storage

The data base should store each data item only once in a hierarchical fashion to reduce data storage. Duplication of data will occur in the data base if it is desirable to make explicit data item links between data set records rather than use record pointers. Some data sets will use ordered sets for fast look-up purposes. This technique is trades speed of access for data storage but is well justified if rapid data retrieval is necessary.

5. Multiple User Access with Security Protection

The data base should be accessible by many users at one time.

TABLE # 2
SAMPLE CHIP HIERARCHICAL DESIGN LEVELS

<u>LEVEL #</u>	<u>POSSIBLE MODULE DESCRIPTION</u>	<u>RANGE OF SUB MODULES/ LEVEL</u>	<u>DEVICES/ SUBMODULE</u>	<u>TYPICAL TOTAL DEVICES/MODULE</u>	<u>TYPICAL NUMBER SUBMODULES/LEVEL</u>	<u>TYPICAL MODULES</u>
1 Highest	Chip (System)	5 - 50	4K - 100K	100K - 2.5M	25	1
2	Blocks (Processor memory)	5 - 20	400 - 10K	4000 - 100K	10	25
3	Functions (Adders, Counters)	5 - 20	20 - 500	400 - 10K	20	10
4	Cells (4 Bit slice, PLA)	5 - 50	2 - 50	20 - 500	10	20
5 Lowest	Primitive (Group of gates)	-	-	2 - 50	-	10

Note: Each lower level represents the submodules of the level above: i.e. Level 1 (Chips) will have typically 5 - 50 submodule blocks.

Most DBMS allow this feature but care must be taken to secure any cell within the data base that is currently being updated by a user. If this is not done, then control over the cells and data items is lost and consistent results cannot be insured.

6. Audit/Recovery

Audit and recovery is a major factor in a production data base to guarantee that data is not lost or destroyed. To implement these features without a software data management system would be very difficult. A reasonable alternative is to copy all files at prescribed intervals for back-up.

7. Update of Data

Each data set record can be assigned a start and stop revision to allow update control. Update control is vital to track a design's history.

4.0 A Logical and Physical Data Model

The key to whether a data base can service a VLSI DA system is how simple, yet general, is its model of the logical and physical system. This section will outline a system model as well as propose a data base structure. Terms related to the data base structure and estimates of the size of the data base records for each data set will also be given.

4.1 System Model

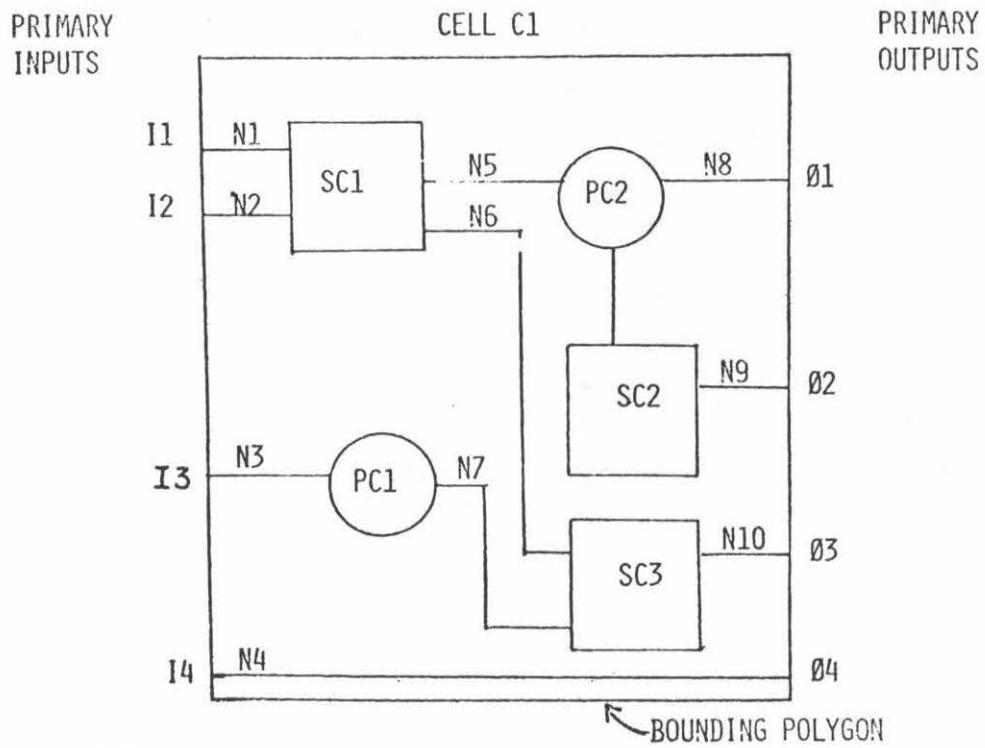
Figure 2 indicates the proposed system model. The basic building block is a cell which is composed of subcells, which themselves can be cells, and primitives which are the bottom or lowest level cells which are not further decomposed.

Each cell is composed of 4 basic parts.

1. Cell Bounding Box (Polygon)
2. Cell primary input and output pins or I/O pins and subcell pins
3. Subcell or primitive references.
4. Net (external and internal) and their associated pins

The primitive has parts 1, 2 and 4 only and does not reference other cells or primitives.

The cell bounding box or polygon is the geometrical enclosure of the physical cell. The cell primary inputs or outputs are the points which join the bounding box to the external nets. The internal nets join the subcell and primitives to other subcells or primitives but do not interface to the external parts of the cell. The subcells are themselves cells which make up the higher level cell. The primitives are the

CELL DATA BASE MODELKEY

CELL - C

SUBCELL - SC

PRIMITIVE CELL - PC

N - NET

I INPUT

Ø OUTPUT

PARTS OF A CELL

1. CELL BOUNDING POLYGON
2. CELL PRIMARY PINS
3. SUB CELL/PRIMITIVE CELLS
4. NETS

Figure 2

lowest level cells which are composed of explicit logic function descriptions and physical geometric descriptions of the primitive. Note that a cell can also be a primitive if it has a logical functional (behavioral) description in terms of its inputs, outputs and states and/or is explicitly defined in terms of its physical geometry.

4.2 Logical and Physical Descriptions

The cell defined in section 4.1 can have both a logical description in terms of its nets (pins and subcells) or its dual model which is subcells (pins and nets) and a physical model which represents the physical geometries of several notations. Each physical description can be thought of as a notational description or representation. Figure 3 indicates 4 possible notations that might be useful to a designer in VLSI work. The logic schematic notation for each of the 4 parts of a cell is shown in terms of its physical descriptions. i.e.: a bounding box, a dot for an I/O pin, a line for a cell net and a geometrical symbol representing the subcells.

Likewise, other notations such as "stick", or electrical have explicit physical descriptions for each piece of a cell. For VLSI work the physical mask notation is most useful as this represents the physical set of rectangles, boxes, and polygons and that make up the geometries of a VLSI mask. Other notations or mixed notations could be capable of being described as long as they contain the 4 basic parts of a cell.

4.3 Our Machine (OM) Hierarchical Description

At this point an example of a hierarchical description of the Caltech, OM data chip is presented. Figure 4 indicates the 4 level hierarchical structure for the "OM" data chip. Shown for each level is the number of cells/level and the estimated number of records/cell. Table 3 indicates an estimate of the total data base records required to store all cells and also indicates the total number of records if all data in all cells was expanded. A 6 to 1 hierarchical data compression ratio results for this "OM" example. Table 4 indicates the estimated number of records for each data set.

5.0 Data Base Structure

Figure 5 outlines a proposed tree data base structure for VLSI DA design. The boxes in the structure represent individual data sets. A data set is a collection of data records in a file usually organized on a set of keys. Index sets, which are collections of records pointing to data set records, are shown as circles and could be applied to each data set for fast look-up purposes. The data base structure is divided into 2 major areas: the logical design and the physical/graphic design. The physical design is complementary to the logical in the sense that each

SAMPLE ENGINEERING NOTATIONS

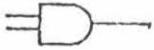
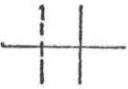
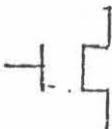
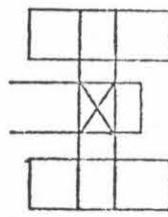
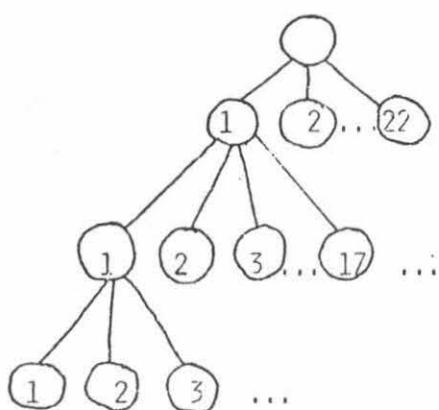
NOTATION TYPE	MODULE BOUNDING BOX	MODULE PINS	MODULE NETS	PRIMITIVES SYMBOLS
LOGIC	Y X RECT.	LOGIC PINS	LINES	
STICK	Y X RECT.	INTERSECTIONS	LINES	
ELECTRICAL	Y X RECT.	DOTS	LINES	
PHYSICAL	Y X POLYGON	CONTACTS	SHEETS	 SET OF SHEETS

Figure 3

FIGURE #4

HIERARCHICAL "OM" CHIP MODEL

CELL DESCRIPTION	LEVEL	USED CELL/LEVELS	RECORDS /CELL
CHIP	LEVEL 1	1	482
SUBFUNCTION	LEVEL 2	22	652
MINOR - FUNCTIONS	LEVEL 3	17/37	353
BASIC CELLS	LEVEL 4	3/150	87

TABLE #3
"OM" DATA BASE RECORDS

LEVEL	TOTAL EXPLODED RECORDS/LEVEL	TOTAL DATA BASE RECORDS
1	(1x 482)	482
2	(22 x 652)	14344
3	(22 x 17 x 353)	132022
4	(22 x 17 x 3 x 87)	97614
	TOTAL	244462
		TOTAL
		40937

DATA COMFACTION RATIO: 6:1

TABLE #4

ESTIMATED RECORDS / LEVEL FOR "OM" CHIP

NUMBER OF LEVELS FOR CHIP >>	4
DIFFERENT SUBMODULES AT LEVEL 1 >>	22
SUBMODULES PER MODULE AT LEVEL 1 >>	22
NETS PER MODULE AT LEVEL 1 >>	100
PINS PER MODULE AT LEVEL 1 >>	250
GT PER MODULE AT LEVEL 1	100
RULES PER MODULE AT LEVEL 1 >>	10
DIFFERENT SUBMODULES AT LEVEL 2 >>	37
SUBMODULES PER MODULE AT LEVEL 2 >>	17
NETS PER MODULE AT LEVEL 2 >>	150
PINS PER MODULE AT LEVEL 2 >>	375
GT PER MODULE AT LEVEL 2 >>	100
RULES PER MODULE AT LEVEL 2 >>	10
DIFFERENT SUBMODULES AT LEVEL 3 >>	150
SUBMODULES PER MODULE AT LEVEL 3 >>	3
NETS PER MODULE AT LEVEL 3 >>	70
PINS PER MODULE AT LEVEL 3 >>	170
GT PER MODULE AT LEVEL 3 >>	100
RULES PER MODULE AT LEVEL 3 >>	10
DIFFERENT SUBMODULES AT LEVEL 4 >>	0
SUBMODULES PER MODULE AT LEVEL 4 >>	0
NETS PER MODULE AT LEVEL 4 >>	10
PINS PER MODULE AT LEVEL 4 >>	27
GT PER MODULE AT LEVEL 4 >>	40
RULES PER MODULE AT LEVEL 4 >>	10

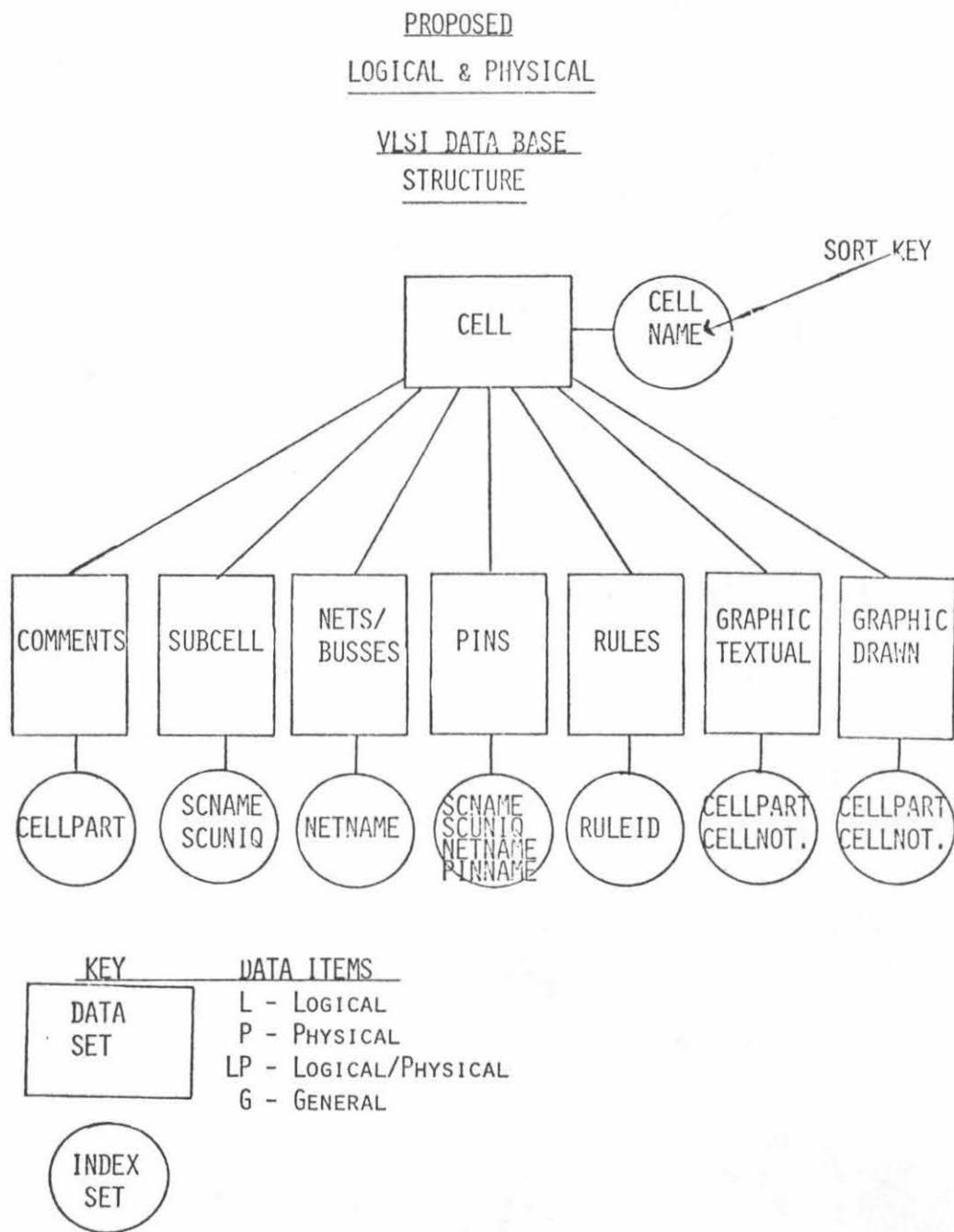


FIGURE 5

logical quantity relative to a cell contains one or more physical descriptions i.e. : nets have physical pad sizes and shapes, etc. Treating each physical quantity in the data base as a part of the notation system, allows multiple physical descriptions for one logical description.

5.1 Cell Data Set

The cell logical data set contains all the global data items, such as number of subcells, in the cell or whether a cell is a primitive. This top level data set is the entry point to each user program in that all subcells which are of interest can be derived in a top down fashion from the highest level cell. Only those levels in the design hierarchy which are of interest need be accessed. For instance, if a simulation of a VLSI chip at its next lowest level is needed, then only 1 level down in the hierarchy need be accessed. Figure 6 indicates a block diagram of the OM data chip. The chip is broken down into 22 subcells. If a logic primitive functional description was written for each of these subcells in terms of its input, output and states, then the chip could be simulated in terms of its 22 pieces. Likewise, if a logic primitive functional description was written for the chip, then this description could be simulated and compared against the simulation of its 22 pieces to verify consistency of the logical design. This technique can be a useful in design verification and the data base can house the logical functional description in the rules data set.

5.2 Subcell Data Set

The subcell logic data set will hold all the references to subcells or primitives within a given cell. Hierarchy is developed by iterating back to the cell level for each subcell to find its associated pieces.

5.3 Comment Text Data Set

This data set is intended to be a user comment text file. Possible uses for this data set could be to store cell specifications at the next higher level, user documentation of the design or design status of the cell.

5.4 Nets/Busses Data Set

The nets/busses logic data set will contain all nets inside a cell. Net types can be external, those touching the cell bounding box or internal, those which do not connect to external I/O pins.

5.5 Pins Data Set

The pins logic data set will house all the pin related data in each cell and the associated explicit pin information for each

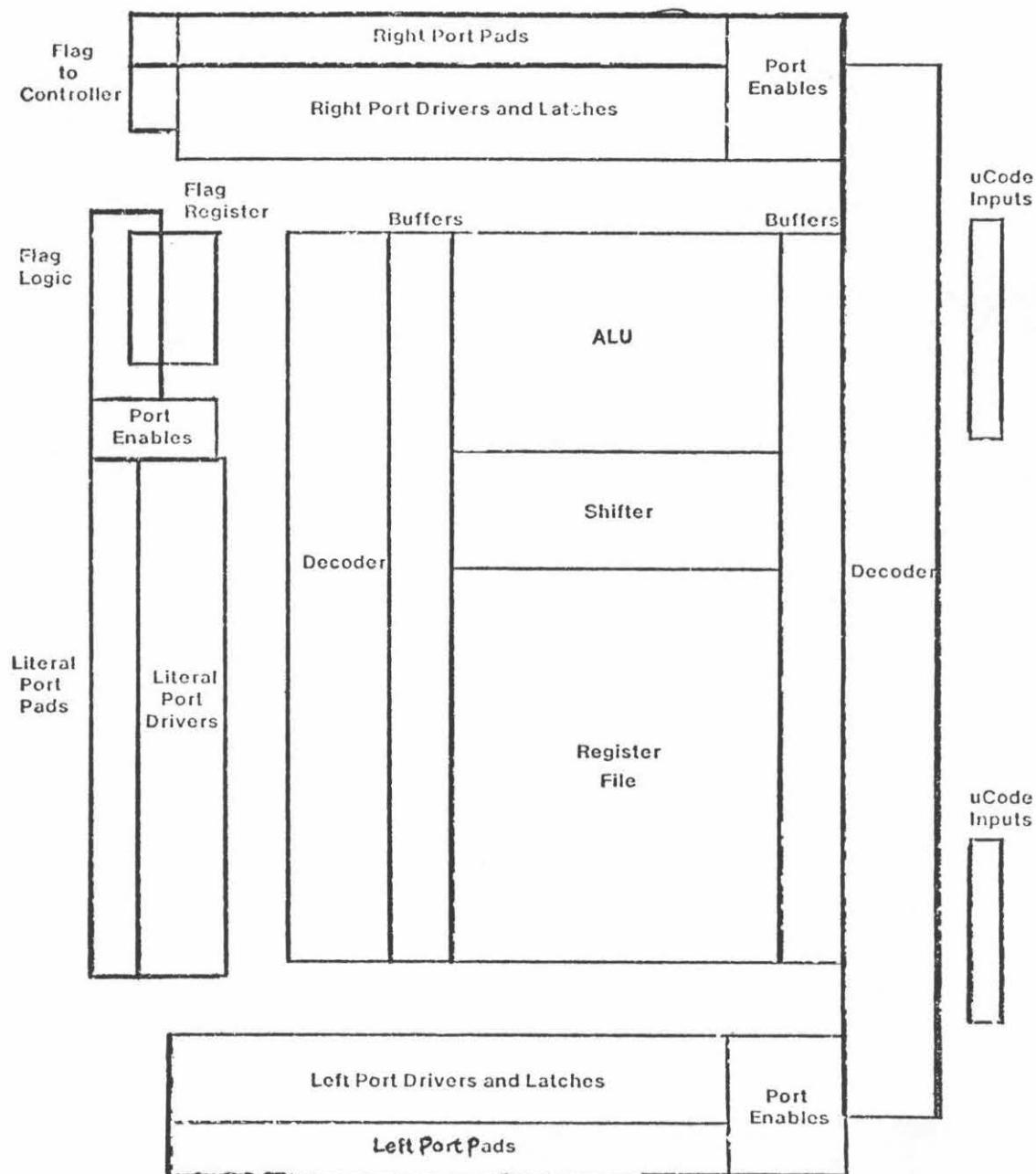


Figure 6 Map Locating the Major Blocks of the Datachip

subcell within a cell.

5.6 Rules Data Set

The rules logical and physical data set would allow a card image description of the rules for each level of notation the user wishes to employ. The language for each of these rules notations can be natural to that notation. For example, the mask physical notation would employ a design rule check language which would outline the design rules for inter and intra layer spacings. The electrical notation would house rules similar to those for circuit simulators such as MSINC or SPICE and could contain exactly those languages. The "stick" notation could contain spacing rules relative to each line in a stick drawing and the logic notation rules could contain the functional description of the primitive in terms of its inputs, outputs and state.

5.7 Graphic Textual Data Set

This data set houses the graphic textual data for one of the 4 physical parts of a cell. A graphic textual language could be standardized across all notation types or could be specific to a notation. Languages similar to CIF [2], and ICLIC [3] could be used to describe the physical geometries of a cell.

5.8 Graphic Drawn Data Set

The graphic drawn data set stores the user drawn data representing one of the 4 physical parts of the cell. A user could sit in front of an interactive storage or refresh tube and physically draw a primitive structure, a net track, a cell bounding box or a cell pin and expect this polygon data to be stored for later use. Figure 7 indicates graphic textual and graphic drawn data. Methods for translating from graphic textual data to graphic drawn data are well known since this is a one-to-many transformation. Translating from graphic drawn to graphic textual is more difficult since this is a many-to-one transformation requiring some form of pattern recognition. Recognition of step and repeat patterns would be very difficult in this reverse translation. The pattern recognition process is needed in VLSI design for determining a transistor or gate from a set of polygons.

5.9 Data Base Size Estimates

One important aspect of the proposed data base is how large will it become for a chip of 100K devices. Tables 5 and 6 relate to the data shown in Table 2 and indicates data set sizes for each of the data set types described. The key concept here is the hierarchical structure of the data base and allowing necessary data to be carried only once in any one cell or primitive.

Assumptions in the calculation of set sizes:

FIGURE 7

GRAPHIC TEXTUAL DESCRIPTION

POLYGON X1 Y1 X2 Y2 X3 Y3 X4 Y4;

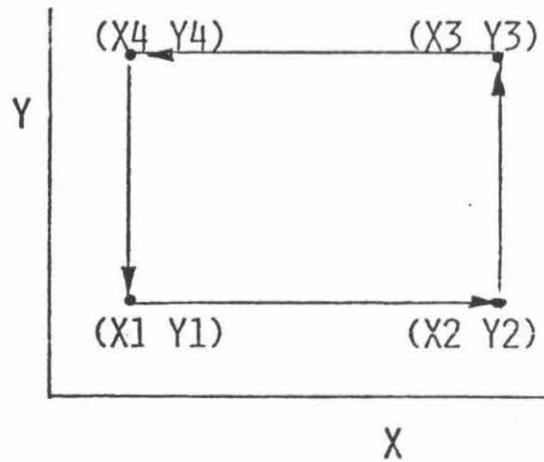
GRAPHIC DRAWN DESCRIPTION

TABLE 5

ESTIMATES OF DATA SET SIZES IN A VLSI DATA BASE*
 FOR 100K DEVICE CHIP

DATA SETS	ESTIMATED RECORD SIZE (CHARS.)	ESTIMATED TOTAL RECORDS FOR 100K DEVICE CHIP	TOTAL CHARS.
CELLS.& PRIMITIVES	189	766	12,474
COMMENT TEXT	98	1320	129,360
NETS	39	13200	514,800
PINS	64	35640	2,280,960
SUBCELLS	53	1320	69,960
RULES	90	4500	405,000
GRAPHIC TEXTUAL	98	8000	784,000
GRAPHIC DRAWN	20	16000	320,000
TOTALS	661	80046	4,516,554 CHARACTERS

* INITIAL ESTIMATES ONLY - NOT DEEMED ACCURATE

TABLE 6
ASSUMED AVERAGE SIZE OF EACH CELL

DATA BASE ITEM	QUANTITIES
COMMENT TEXT	20 LINES
NETS/BUSSES	200
PINS	$2.5/\text{NETS} + 40/\text{CELL} = 540$
SUBCELLS	20/CELL
RULES	200 LINES/PRIMITIVE, 20 LINES/MODUL
GRAPHIC TEXTUAL	500 LINES/PRIMITIVE, 100 LINES/MODUL
GRAPHIC DRAWN	1000 LINES/PRIMITIVE, 200 LINES/MODUL

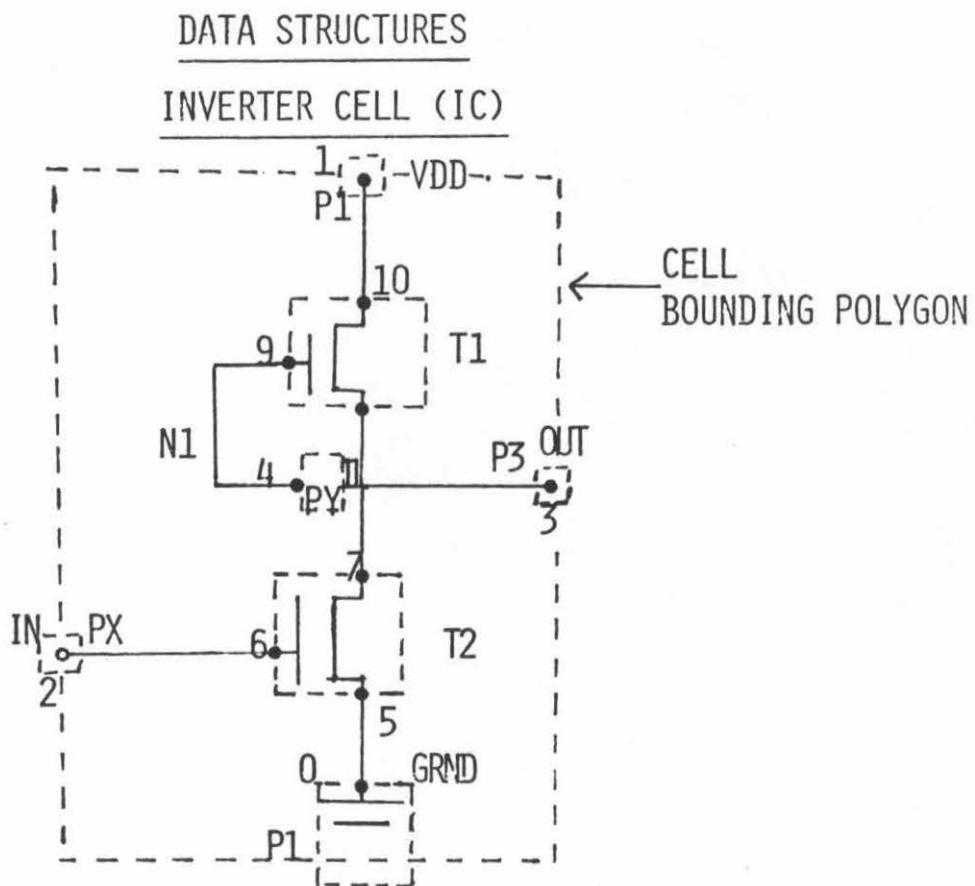
INVERTER ELECTRICAL DRAWING

FIGURE 8

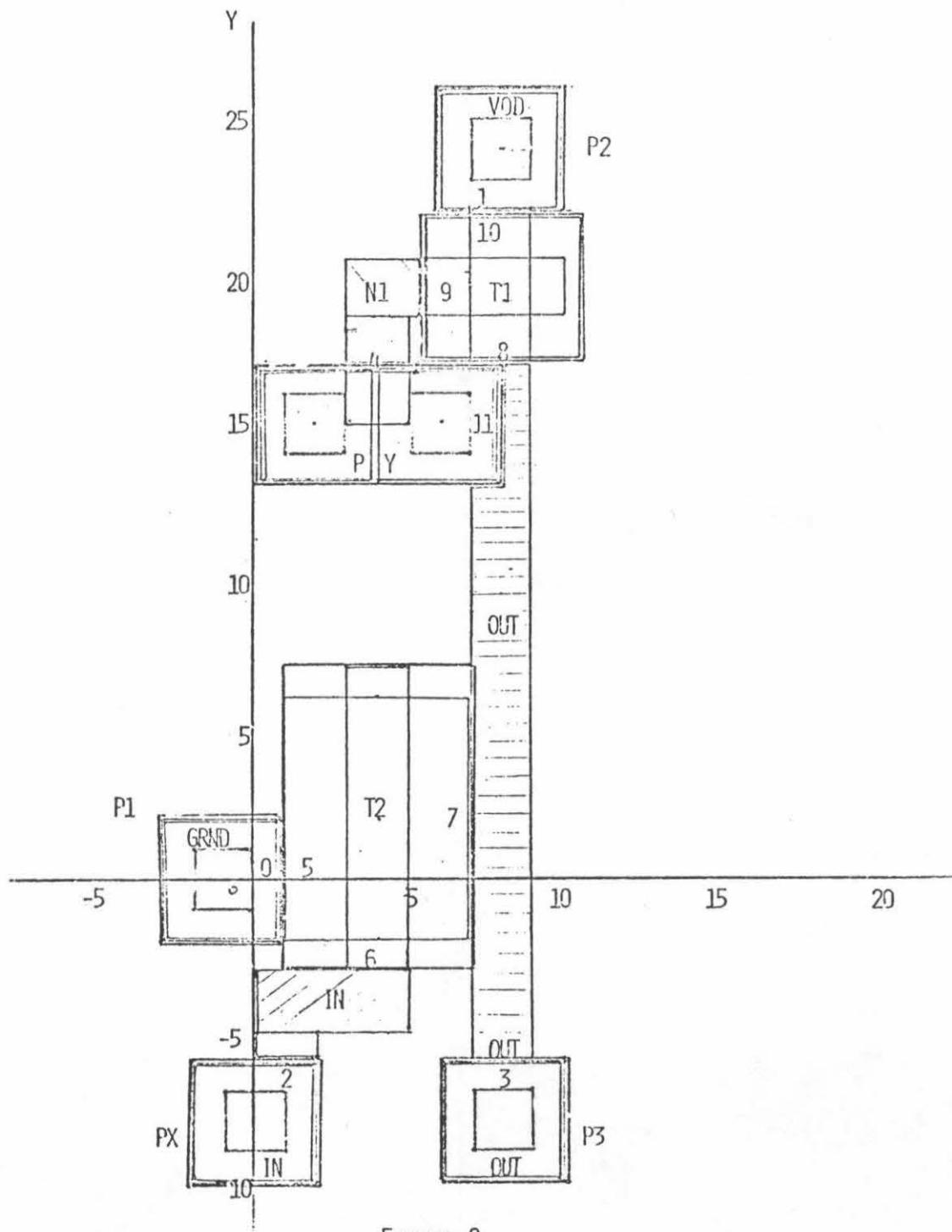


FIGURE 9

INVERTER CELLDATA BASE DATA

DATA SET	DATA			
CELL	CELLNAME (INVERTER CELL - IC) INPUTS (1,2) OUTPUTS (0,3) PRIMITIVE (NO)			
SUBCELLS	SUBCELLNAME	SUBCELL UNIQ	SUBCELL LOG	COMMENT
	P	1	-3, -2	PAD
	P	2	6, 22	PAD
	P	3	6,-10	PAD
	PX	1	-2,-10	PAD
	PY	1	0, 13	PAD-DOUBLE
	T1	1	5.5, 17	DEPL TRAN
	T2	1	1, -3	ENH TRAN

TABLE 7

INVERTER CELLDATA BASE DATA

DATA SET	DATA			
NETS	NETNAME	NETCLASS	# PINS	PINS
	VDD	POWER	2	1,10
	OUT	EXTERNAL LOGIC	4	3,7,8,11
	IN	LOGIC	2	2,6
	GRND	POWER	2	0,5
	N1	INTERNAL LOGIC	2	9,4
PINS	CELLNAME	UNIQ	NETNAME	PINID
	IC	1	GRND	0
	IC	1	VDD	1
	IC	1	IN	2
	IC	1	OUT	3
	T1	1	VDD	10
	T1	1	N1	9
	T1	1	OUT	8
	T2	1	OUT	7
	T2	1	IN	6
	T2	1	GRND	5
	P	1	GRND	0
	P	2	VDD	1
	P	3	OUT	3
	PX	1	IN	2
	PY	1	N1	4
	PY	1	OUT	11

TABLE 8

INVERTER CELLDATA BASE DATA

DATA SET	DATA		
	CELL NOTATION	CELL PART	TEXT
GRAPHIC TEXT	PHYSICAL POLYGONS (MASK)	CELL BOUNDING POLYGON (1)	POLYGON ((-2,-10), (2,-10)...);
		CELL PRIMARY PINS (2)	CALL P1; CALL P2; CALL P3; CALL PX;
		↓ SUBCELLS (3)	CALL PY; CALL T1; CALL T2;
		↓ NETS (4)	VDD - NONE OUT - POLYGON ((7,-6), (9,-6), (9,17), (8,17), (8,14), (7,14), (7,-6)); IN - POLYGON ((0,-6), (2,-6), (2,-5), (5,-5), (5,-3), (0,-3), (0,-6)); GRND - (NONE) N1 - POLYGON((3,18), (5.5,18), (5.5,21), (3,21), (3,18));
		↓	

TABLE 9

INVERTER CELLDATA BASE DATA

DATA SET	DATA	
RULES	RULE ID	RULES
	1. FUNCTIONAL SIMULATION	OUT = \overline{IN} ; DELAY = IONS;
	2. MSINC ELECTRICAL SIMULATION	T1 = BDEP; 1λ , 1λ ; T2 = BENH; 4λ , 1λ ; TOL = .01 VDD = 5VOLTS; VIN = 5,400N,0, 410N; TIME = 10N, 300N; PLOT = VOUT (0,3), VIN (0,2);
	3. DESIGN RULE CHECK	DIFFUSION = 2λ ; POLY = 2λ ; POLY TO DIFFUSION = 1λ ; METAL = 3λ METAL TO METAL = 3λ ; ETC. $\lambda = 2$
	4. DIMENSIONAL ETC.	

TABLE 10

1. Only 5 levels in hierarchy for sample hypothetical chip.
2. Primitives composed of 2 - 50 transistors
3. Number of typical subcells at each level in hierarchy as shown in Table 2.

Table 5 points out that over 4 1/2 million characters of storage would be needed for a chip of 100K devices.

6.0 Sample Data Mapping of an Inverter Cell into Proposed Data Base

Figures 8 and 9 indicate simple inverter cell electrical and mask drawings. Each subcell and node is numbered to allow a net list to be constructed.

Tables 7-10 indicate the outline of the data base data for each selected data set as described earlier.

It would appear that a simple data base model and structure as described, used in a hierarchical fashion, would suffice for a Design Automation System.

References

- [1] Computing Surveys, Volume 8, 1976, ACM
- [2] Introduction to VLSI Systems, C.A. Mead, L.A. Conway
Chapter 4, Section 2, To be published
- [3] A Language Processor and a Sample Language
Thesis - R. Ayres, June 12, 1978, California Institute of Technology

Bristle Blocks: A Silicon Compiler

Dave Johannsen
Caltech

Abstract

Standard LSI Design Automation systems are database management systems that aid the circuit designer by organizing the collection of submodules that comprise a chip. This type of file system usually does not aid in the actual computation of silicon layout, and can hinder a designer with program constraints that have little or nothing to do with silicon constraints. The Bristle Block system is an attempt to create a silicon compiler that will perform the majority of the implementation computation while placing a minimum set of constraints on the designer. The goal of the Bristle Block system is to produce an entire LSI mask set from a single page, high level description of the integrated circuit.

Introduction

After designing a few reasonable sized LSI integrated circuits, it becomes painfully obvious that the current design tools aid little in the every increasing task of automating chip design. Most systems are little more than fancy filing cabinets that help organize the data associated with various elements composing a chip. Even with these advanced systems, 90% of the design is completed in 10% of the time, while the remaining 10% of the design requires the remaining 90% of the time. Too many design decisions are "frozen in concrete" early in the game, before the effects of those decisions become apparent.

What if a person were able to sit down and design a complete chip in a single afternoon? The user would grab a few building blocks, snap them together, and experiment with many radically differing designs before deciding upon the actual implementation. What if that person were given complete mask layouts and simulations for each of his or her experimental configurations with almost no effort? This is the environment that the Bristle Blocks system attempts to provide.

Design Goals

The goal of the Bristle Block system is to allow the user to design LSI integrated circuits with as little concern for the mechanics involved as possible. The system must be capable of designing fairly complicated chips, and the resulting designs should have a high degree of optimization in the layout. The results should be competitive with "hand layout" chips.

Structured Design. The Bristle Block system is intimately dependant on the structured design methodology, as presented in Mead and Conway [1]. This system encourages the design of regular computing structures. These regular computing structures tend not to limit the power or usefulness of the resulting machines, but rather to enhance the performance and generality of the hardware.

Hierarchical Design. The chips designed by Bristle Blocks are hierarchical in nature. This hierarchy imposes a locality on the various sections of the chip that can be exploited when performing design rule verifications and electrical simulations of the chip.

Various Representations. The Bristle Block system provides various representations for the integrated circuit. The representations span the entire range from the physical to the conceptual aspects of the chip. Bristle Blocks is designed to handle the following seven representations:

LAYOUT. The "lowest" level of representation is the Layout level, which produces the actual chip masks.

STICKS. The Sticks level produces a *stick diagram* of the chip, which has the same topology as the layout, but with all of the features reduced to single-width lines. The resulting diagram is much easier to comprehend than the full layout diagram.

TRANSISTORS. The Transistor level produces a transistor diagram for the chip or subsection of the chip.

LOGIC. The Logic level of the chip can produce a logic diagram of the chip in the TTL style.

TEXT. The Text level prints a hierarchical description of the chip that can be used as a "user's manual" for the completed chip.

SIMULATION. The Simulation level can be used to logically simulate the chip, so that software can be written for the chip to explore the feasability of the design.

BLOCK. The Block level draws a block diagram of the chip, showing the arrangement of the buses and core elements.

Every fundamental element in the Bristle Block system has the capability of containing each of these seven representations for itself. When the system is physically connecting the elements, the logical and textual links are also generated. Other representations may be added to the system at any time.

Bristles and Blocks

The fundamental unit in the Bristle Block system is the cell, which may contain geometrical primitives and references to other cells. These cells to the LSI designer can be equated to the programmer's subroutines: each contain a few primitive operations and references to lower levels in the hierarchy. The geometric primitives are instances of lines, boxes, and polygons, each with an associated mask layer.

Bristle Blocks also has a structure for specifying the location and flavor of connection points between cells. These connection points are like bristles along the edges of the cells, and it is upon these bristles that the Bristle Block system builds most of the computable structures. Connection points help keep local data local and global data global, while delaying the binding of many design constraints. For instance, a cell that requires an input from a pad would contain a connection point stating where in the cell the pad should connect and what type of pad is needed. When the chip is compiled, the appropriate pad is automatically placed on the chip and a wire is routed between the pad and the cell. Thus, the local data, which declare where in the cell the pad connects and the type of pad, is kept local to the cell, while the global data, which specify where the pad is located and how the wire is routed, is kept global to the cell. Also, none of the data need be supplied by the user at compile time.

The data necessary to specify the various representations for the cells and connection points may be stored in disk files and read in as needed, to allow for the use of common cell libraries and sharing of data. Any low level cell that the user may need for his chip must be entered into either the system or a library before the chip can be compiled. Associated with each cell is the information necessary to extract the cell's definition from a file, if such a file exists.

The low level cells in a library are defined by entering the actual layout of each cell representation in a standard cell design language. The low level cell design task was not given to the compiler, but left to the user, for three reasons:

- 1) The design of low level cells does not take much time. Each cell usually has a small area, few transistors, and is reasonably well defined.
- 2) Very few mistakes are made in the design of low level cells. Experience has shown that virtually none of the fatal chip errors occur in the low level cells, but that chips fail because of faulty "glue": the high level cells holding the chip together.

3) Human ingenuity pays off well in the low level cell design. There has yet to be written a computer program that approaches human ingenuity in cell design, which is most noticeable in the low level cells.

Chip Format

As a starting point, one style of chip design was selected to explore the possibilities of a Bristle Block system. Although it was initially felt that this style would be rigid and special purpose, the generality of the style became apparent as the potentials of Bristle Blocks were discovered. The style of chip design is stated in the description of three formats of the chips: the physical, logical, and temporal formats.

Physical Format. The physical format of chips built by the current Bristle Block system is shown in figure 1. A chip consists of a central *core* which is controlled by an *instruction decoder*, both of which are surrounded by pads. The core is composed of a series of data processing elements, such as memories, shifters, and arithmetic-logic units. Based upon the requirements of the core, the instruction decoder and pads are automatically generated and placed in the final chip.

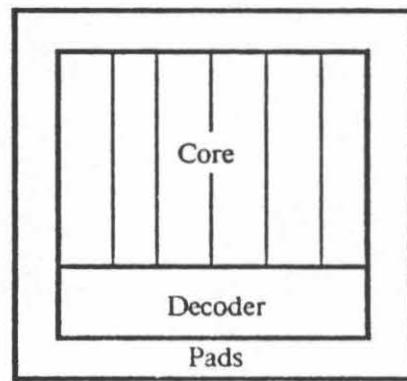


Figure 1. Physical Chip Format

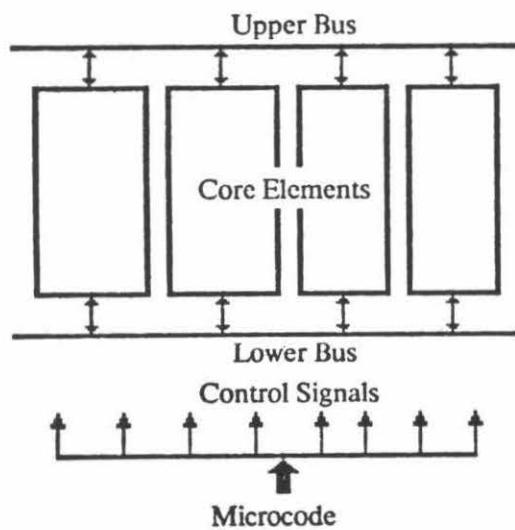


Figure 2. Logical Chip Format

Logical Format. The logical format of the chips is shown in figure 2. Each of the core elements can communicate with either of two buses that run through the elements. These buses may run the length of the chip, or they may stop anywhere along the chip with new buses servicing the remainder of the chip. The order of placement of elements along the core is irrelevant to the system, with the exception that at most two buses may run through any element. The microcode words which control the operation of the chip enter the decoder twice during each clock cycle. The appropriate control functions are derived from these microcode words and latched by control signal buffers. These buffers then drive the control lines of each core element.

Temporal Format. The chip is driven by a two phase, non-overlapping clock. One phase, referred to as φ_1 , controls the transfer of data between elements via the buses. The alternate clock phase, referred to as φ_2 , controls the operation of the data processing elements. During φ_2 , the buses are precharged to a high state. Any processing element that is activated during φ_1 may pull the bus lines low. Any element may read the data from the bus by the end of φ_1 . While the buses are transferring data, the data processing elements may be precharged. An example of a precharging processing element is an adder, where the carry chain is precharged to a high state. During the following φ_2 , the various points along the carry chain may be pulled to a low state. Instructions enter the control buffers through the decoder logic on the clock phase preceding the phase when the instruction is to be executed.

Operation

Bristle Blocks is a three pass compiler, consisting of a core pass, a control pass, and a pad pass. The core pass takes both the user's input and low level cell definitions to construct the core of the machine. The control pass adds the instruction decoder to the core by generating a decoder which fills the constraints posed by *control connection points* in the core. The pad pass adds the pads to the perimeter of the chip and routes wires between the pads and the corresponding connection points in the core and decoder.

User Input

The input to the compiler consists of three sections. The first section states the microcode instruction width and describes the decomposition of the microcode word into various fields, such as the "Register Select Field" or the "ALU Operation Field." The second section states the data word width for the chip and lists the buses that run through the core of the chip. The final section lists the elements of the chip's core, and provides any parameter values that the elements require.

Pass 1: Core Layout

The core pass is driven by the user's input. The input contains the list of elements and associated parameters. After all of the elements vote on the values of global parameters, each element is executed in turn, resulting in a hierarchy of cells which implement the core of the chip.

To allow any two elements to plug together, cells must interface either by maintaining a common pitch (width) or by wire routing. To save the space and costly routing needed if cell widths vary, a design constraint states that all cells must be of equal width. To have a uniform width for all cells, every cell must be *designed* as wide as the widest cell. The width of the widest cell is not known until *after* all of the cells are designed! And as future cells are designed, they must either be forced to have the same width as current cells, or else all of the cells must be redesigned to accommodate the wider cells.

By introducing stretchable cells, this problem can be avoided. Each of the cells are designed with places to stretch. As the core is being scanned, each element reports the width of its cells, so that when the end of the core list is reached, the widest cell width is known. As the elements produce their cells, each cell is stretched (a painless operation) to fit all other cells. The cells can also be stretched to allow the power lines to expand as power demands increase.

Another restriction arising from the arbitrary element ordering is that each of the cells must meet certain interface requirements. Proper interface standards eliminate intercell problems such as shorting out a neighbor's transistor or power supply line. By agreeing on a standard interface to begin with, any cell can be guaranteed to mesh properly with adjacent cells before the neighboring cells are specified. Boundary conditions like these allow design rule checking to be performed on individual cells as the cells are designed, rather than on fully instantiated artwork.

Along with meshing of cells, the busing requirements between cells must also be met. Buses may need to break or stop, and bus precharge circuits must be added for each bus. Details like these need not be specified by the user, but are added by the compiler.

Pass 2: Control Design

Given the results of the core pass, the control design and layout proceeds. First, control buffers to drive the control lines are inserted along the edge of the core. The timing is also added to the control signals by the buffers. Then, a text array is constructed which specifies the decode functions needed for each buffer. A two-tape Turing machine operates on one "tape", which contains the text array, and writes the second "tape", producing compiled silicon code. When it has finished operating on the array, the Turing machine will have generated and optimized the instruction decoder, and created pad connections for the inputs to the decoder.

Pass 3: Pad Layout

The pad layout pass of the silicon compiler begins by collecting all of the connection points which need to be connected to pads. These connection points are sorted in clockwise order, and pads are allocated in the same order. The pads and connection points are examined by a Roto-Router, which rotates the pads around the perimeter of the chip in an attempt to minimize the length of wire between pads and connection points. The Roto-Router spaces the pads evenly around the chip to avoid generating pad layouts that would be difficult to bond. The third pass concludes by adding wires between the pads and the connection points.

Conditional Assembly

Bristle Blocks can allow for conditional assembly of silicon. For example, when designing prototype chips, the internal state of a state machine may need to be routed to pads, but when production chips are produced, the area of the pad and wires may need to be reclaimed. The user may declare

a global boolean variable PROTOTYPE, which, if TRUE, will add the connection points for the pads, but if FALSE will not. At any time prior to actually compiling the chip, the user may decide whether this is a prototype chip or not, and properly modify the layout.

Since Bristle Blocks provides a high level programming language to the user, cells can be smart, and perform calculations as they are being added to the core. As an example, there may be several possible cell layouts which implement the desired function. After the cell width has been determined, the possible layouts which fit within the specified width can be judged to find the cell with minimum resulting area. The minimum area cell is then used in the actual chip, which optimizes the length of the chip.

Implementation

Bristle Blocks is implemented in ICL, a high level graphics, integrated circuit design, and general purpose programming language created by Ron Ayres [2]. ICL allows the user to generate new data structures at any time. Functions and coercion can also be added to manipulate the data structures. The coercions are invoked automatically by ICL when the data it receives are not of the type it expects. Also, ICL provides for polymorphic functions, where the operation performed by the function is dependent upon the data supplied to the function. ICL is an extensible language, so that any user defined functions, datatypes, and coercion become a part of the current version of ICL. This also means that the Bristle Block system provides the user with all of the basic ICL operations, and that Bristle Blocks is an extensible system..

The only disadvantage with using ICL is that the current version is an implementation for a PDP10/PDP20, which has a small (18 bit) address space.

Conclusion

In the 2½ man-months that have gone into developing the first Bristle Blocks system, approximately 80% of the system has been implemented. Given a high level description of the chip and definitions for core elements, the system produces a complete layout, sticks diagram, transistor diagram, logic diagram, and block diagram. There are a few special cases in the circuit topology that have still to be considered, but lack of time is the only reason they have not been implemented. Hooks for the circuit simulator and text producing code have been included in the system, but with the constraint that all code must remain in core, these features must wait.

The chips produced by the system are fairly well optimized, having $\pm 10\%$ of the area of a chip produced by hand using the structured design methodology. The compiler takes approximately 4 minutes to generate a small chip, in all five of the current representations. The time needed to generate a fairly large chip should be in the neighborhood of 10-15 minutes.

At present, the Bristle Blocks system is tailored to produce chips of a particular architecture. As the domain of silicon compilers is examined through the use of the current Bristle Blocks, a more general chip architecture will be found that encompasses the current chip architecture. A more generalized Bristle Blocks will be developed to compile the generalized chip architecture. The generalization of the current Bristle Block system may continue through a number of generations. At some point, however, a totally new Bristle Blocks will be generated to handle a completely different style of chip. Thus, there will emerge several classes of Bristle Block systems, each compiling a separate class of chip architectures, rather than one Super Bristle Block system which attempts to produce *all* chips.

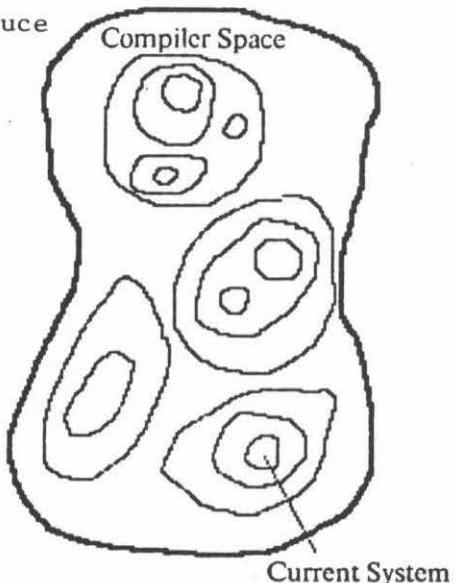


Figure 3. Hierarchy of Systems

References

- [1] Mead, Carver A., and Conway, Lynn "An Introduction to VLSI Systems," limited printing, 1978
- [2] Ayres, Ronald, "ICL Reference Manual," SSP File #1364, CalTech, 1978

Silicon Compilation - A Hierarchical Use of PLAs

Ron Ayres November 1978

1.0 Introduction

This paper proposes a way to *compile* a silicon layout directly from synchronous logic specification. The motivation for introducing *compilation* into the silicon world comes from its extreme success in the software world. As we see silicon area increasing and circuit complexity increasing, we might feel much in common with the early day programmers who faced increasing memory availability along with increasing program complexity.

Software and hardware revolve around the same basic concern: The *software* designer lays out a one dimensional array of *memory* whereas the integrated circuit designer lays out a two dimensional area of *silicon*. In each case, various constraints must be satisfied in order to obtain a working product. In addition, both efforts involve lots of modification.

Compilers are around to reduce the amount of specification required to obtain a working program. Custom programs can be *created* economically and confidently with the aid of compilers and programs can be *modified* quickly even though the resulting memory layout has to change dramatically. Producing custom VLSI requires these same conveniences. Where a software compiler revises addresses which reference relocated blocks of memory, a silicon compiler reroutes wires to relocated blocks of silicon. When changes are made to the logic specification for the chip, the compiler creates a new layout with all cells positioned and connected especially for the new logic specification.

This paper is divided as follows: First, synchronous logic specification and a silicon implementation for synchronous logic is described. Then, problems arising with large systems concerning both their specification and silicon implementation are cited and a solution is presented which both eases large synchronous logic specs and decreases silicon area. This particular solution introduces a flexible hierarchy into the specification of logic and accounts automatically for placement and wire routing. The ideas presented here are implemented currently as programs written in the language ICL which runs on a PDP-10.

2.0 Logic Specification for ICs

Synchronous logic is a language in which one specifies the function of an IC. In almost all IC designs, there exists at some time a synchronous logic specification. The synchronous logic spec provides means for simulating the IC before more detailed work proceeds. Synchronous logic is by no means

the most convenient high level language in which to describe ICs but it does provide a solid starting point. Other high level specification languages can translate their input into synchronous logic.

The language of synchronous logic admits any set of equations written in terms of boolean expressions and variables. In addition, synchronous logic admits the specification of a unit delay in time. For example, figure 1 shows the synchronous logic for a single bit of a resetable counter.

Any set of synchronous logic equations can be implemented in silicon via a PLA (Programmable Logic Array) coupled with some inverters and unit delay flipflops. Figure 2 shows a PLA which implements the counter-bit. A PLA by itself implements any set of logic equations (without delays) which can be written in terms of an OR of AND terms *devoid* of logical NOTs. Attach inverters to the inputs of a PLA and one can implement any set of logic equations. Finally, by adding unit delay flipflops onto the outputs, one can implement any set of synchronous logic equations. The unit delay flipflops are placed on the outputs of those equations which were specified with the $=_{next}$ as opposed to the $=$ by itself.

3.0 Hierarchical or Functional Logic Specification

Conceptual design of any IC involves partitioning the desired function into smaller functional units. The smaller functional units themselves may be partitioned *etc.* so that each of the smallest units can be implemented with confidence. For example, a frequency divider may be divided into one counter + one register. The counter can be decomposed further by defining an individual bit position for the counter.

Functional partitioning provides means to localize design concerns. Functional partitioning also alleviates the need to replicate equations for replicated components, e.g., the equations for one bit in a counter may be written once but referenced as a whole many times to form a full counter. Functional partitioning is old hat in software where programs are written in languages which support function definition and invocation.

Logic specification can be partitioned by introducing the concept of a *logic cell*. A logic cell appears to the outside world as a block box + and interface. An interface is a set of *named signals*. Logic cells are related together by writing equations which involve *signals*, each of which is specified by naming both a particular logic cell and a particular signal in that logic cell's interface. For example, figure 3 presents a definition of an N-bit counter in terms of logic cells each of which represents a single counter bit. The notation \S (looks like *apostrophe s*) is used to retrieve a signal given a logic cell and a name. Figure 4 is a copy of figure 3 with explanations superimposed.

The notation used in figure 3 is working program text which has been used to generate the illustrations for this article. The program text uses the name LOGIC_LEVEL as a synonym for *logic cell*. (LOGIC_LEVEL was intended to mean one LEVEL of LOGIC in a hierarchy).

A logic cell is a datastructure consisting of three fields:

EXTERNALS: *two sets of named pins (input and output)*

RELATIONS: *A set of synchronous logic equations which relate the inputs and outputs of both this logic cell and all of the subcells (GUTS).*

GUTS: *The set of subcells referenced by this cell.*

This hierarchical synchronous logic specification language has been used successfully to define an IC whose function is to drive an SCR to control the brightness of a lightbulb, parameterized by four kinds of instructions. The IC consists of two 6-bit registers, a 6-bit and a 2-bit shift register, a 6-bit and a 3-bit counter, and a flipflop. The logic specification was easy to write and it has been simulated successfully.

4.0 Hierarchical PLAs to Implement Hierarchical Logic Specification

To simulate the hierarchical logic specification, a program smashes the hierarchy and creates one long list of equations. A standard synchronous logic simulator takes it from there. Likewise, a single, giant PLA can be constructed automatically from the long list of equations and therefore provide a silicon implementation.

Unfortunately, a PLA can take an inappropriately large amount of area; the area equals approximately the product of the number of input terms and the number of AND terms. PLA area can be saved when the designer notices that his PLA implements actually several *independent* sets of equations. He can substitute the one large PLA with several smaller PLAs. In the best case, the designer can cut PLA area by a factor of k by implementing k independent sets of equations as k small PLAs instead of one large PLA.

Rather than removing the hierarchy from the logic specification prior to PLA generation, we can let the hierarchy *work for us*. Figure 5 shows a PLA which implements a 16-bit counter and figure 6 shows a hierarchical use of PLAs to implement the same. The subPLAs, the PLAs generated by each of the subcells, are lined up with all their inputs and outputs facing upwards. The RELATIONS of the current logic cell are themselves translated to a PLA and placed on the righthand side of the picture. Finally, wires are placed horizontally above the subcells. These wires transmit signals between the RELATIONS PLA and each of the subcells and the connection points (EXTERNALS) of this logic cell.

This hierarchical use of PLAs as the following principle working in its favor: Local signals are represented locally in silicon. That is, signals relevant only to a subcell remain inside that subcell and do not enter PLAs of other subcells or enclosing cells. In addition, a good functional partitioning minimizes the number of input and output signals. (Notice that software functions generally take in and produce small numbers of parameters). Therefore, the RELATIONS PLA will have in general a minimal number of input and output lines, and therefore a nearly minimal area. With this setup, PLAs will *always* relate inputs and outputs of functional units.

The shape of a layout will depend not only upon the specific logic equations, but also upon the chosen hierarchy. Figure 7 shows another layout for the 16-bit counter which differs only in its functional partitioning. The hierarchy for figure 7 is one level deeper; it consists of four subcells where each subcell itself is a four-bit counter generated by calling COUNTER(4). The RELATIONS for the new top level connect the four counters in series. Figure 8 shows still another 16-bit counter; this hierarchy is five levels deep and each level relates exactly two instances of the level immediately underneath. Even though the different layouts have differing areas, a layout not having minimal area might be chosen merely because it has the right shape for a slot in a larger chip. This program makes no such choices, however. The user can modify his hierarchy to obtain the shape he wants.

The layout for a logic cell always comes out with its interface on the lefthand edge. A cell is utilized as a subcell by rotating it 90 degrees so as to get its interface facing upwards.

This program provides interconnect between the PLAs it generates. The interconnect generator accepts two sets of fixed pins among which it will provide interconnect. These two sets of fixed pins lie horizontally on the bottom and vertically on the right, e.g., the interface pins of the subcells on one hand and the pins of the RELATIONS PLA on the other. In addition, the interconnect generator accepts a set of movable pins which will reside on the left and which will be the interface for this level in the hierarchy. The interconnect generator fixes the positions of the movable pins as it creates horizontal and vertical wires between the two sets of fixed pins and the one set of movable pins.

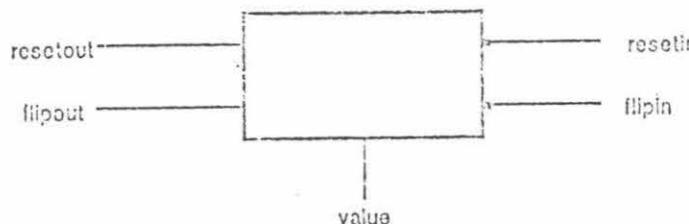
5.0 Some Optimizations

The reader may notice that with the counter shown above, the RELATIONS PLA is trivial, i.e., it represents no logic computation. This RELATIONS PLA serves merely to route signals between the subcells and the counter's interface. The equations in the RELATIONS component of the counter specification contain no logic in fact; they all have the form $A = B$.

Figures 9, 10, and 11 are copies of figures 6 thru 8 where the *trivial* equations are removed. That is, equations of the form $A=B$ have been removed and the signal A has been *indirected* to the signal B so that whenever signal A shows up in a computation, signal B appears instead.

6.0 Conclusions

The continual increase in silicon area invites compilation because there is some space for overhead and because our bigger circuits are getting complex enough to *require* computer assistance like that required by large software systems. Modifications will need to be made more readily and with confident results. In addition, even if silicon compilation is doomed to require too much area, there are still lots of smaller custom ICs which are needed in short order.



```

resetout = resetin
flipout = flipin & value
valuenext = ( IF flipin THEN ~value ELSE value ) & ~resetin

```

COUNTER-BIT

```

DEFINE COUNTER_BIT= LOGIC_LEVEL: BEGIN VAR RESET_IN,RESET_OUT,FLIP_IN,FLIP_OUT,VALUE=PIN;

DO    RESET_IN:= NEW_BIT;
      RESET_OUT:= NEW_BIT;
      FLIP_IN:= NEW_BIT;
      FLIP_OUT:= NEW_BIT;
      VALUE:= NEW_BIT;
GIVE  EXTERNALS: [IN_PIN: { RESET_IN \NAMED 'RESET_IN';
                           FLIP_IN \NAMED 'FLIP_IN' }

                  OUT_PINS: { RESET_OUT \NAMED 'RESET_OUT';
                               FLIP_OUT \NAMED 'FLIP_OUT';
                               VALUE   \NAMED 'VALUE' } ]

RELATIONS: { { RESET_OUT \EQU RESET_IN;
                FLIP_OUT \EQU (FLIP_IN \AND VALUE);
                VALUE   \NEXT CIF: FLIP_IN THEN: NOT(VALUE) ELSE:VALUE \AND NOT(RESET_IN)
                END   ENDEFN
}
(Equations) -->

```

ICL TEXT

FIGURE 1

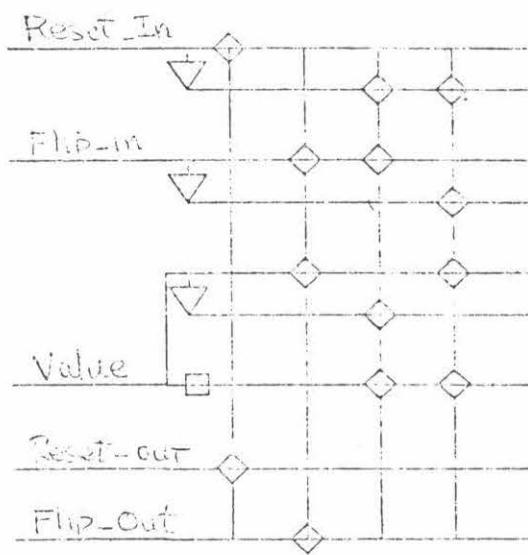
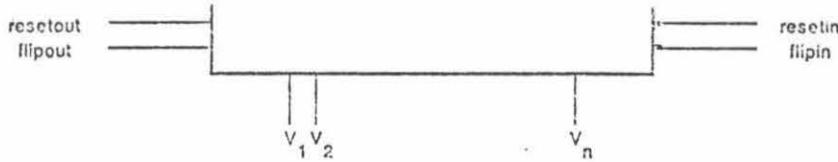


FIGURE 2



```

LET X = COUNTER-BIT [ 1 .. N ]
    resetin of X[i] = resetout of X[i+1]      i = 1 to n - 1
    flipin of X[i] = flipout of X[i+1]

    resetin of X[n] = resetin
    flipin of X[n] = flipin

    resetout = resetout of X[1]
    flipout = flipout of X[1]

    V[i] = value of X[i]   for i = 1 to n
    COUNTER

.

DEFINE COUNTER(N:INT)= .LOGIC_LEVEL: BEGIN VAR CBITS=NAMED_LOGIC_LEVELS; CBIT,LEFT,RIGHT,L,R=LOGIC_LEVEL; I=INT;

DO     CBITS:= {COLLECT COUNTER_BIT \NAMED ('BIT' \SUB I) FOR I FROM 1 TO N;};
       LEFT:= CBITS[1];           RIGHT:= CBITS[N];

GIVE   [EXTERNALS:   [IN_PINS:          {           RIGHT\$ 'RESET_IN';
                                         RIGHT\$ 'FLIP_IN' } }

                  OUT_PINS: {           LEFT\$ 'RESET_OUT';
                                         LEFT\$ 'FLIP_OUT';
                                         CBITS\$S 'VALUE' } ]

RELATIONS:      FOR {L;R} \$CBITS;      COLLECT
                { L\$ 'RESET_IN'      \EQU (R\$ 'RESET_OUT');
                  L\$ 'FLIP_IN'      \EQU (R\$ 'FLIP_OUT') } }

GUTS:      CBITS ]      END      ENDDEFN
            ICL TEXT

```

FIGURE 3

Function Header •

COUNTER

```

  DEFINE COUNTER(N:INT)= LOGIC_LEVEL: BEGIN VAR CBITS=NAMED_LOGIC_LEVELS; CBIT,LEFT,RIGHT,L,R= LOGIC_LEVEL; I=INT;
    DO      CBITS:= {COLLECT COUNTER_BIT\NAMED ("BIT" \SUB I) FOR I FROM 1 TO N;};
    LEFT:= CBITS[1];          RIGHT:= CBITS[N];
  } (For naming convenience,
    let LEFT & RIGHT refer to the
    leftmost & rightmost
    bits.
```

Resulting LOGIC_LEVEL

GIVE . EXTERNALS: {
 The INTERFACE IN_PINS: {
 RIGHT\\$'RESET_IN';
 RIGHT\\$'FLIP_IN';
 }
 OUT_PINS: {
 LEFT\\$'RESET_OUT';
 LEFT\\$'FLIP_OUT';
 }
 CBITS\\$\\$'VALUE' }]

This compact expression produces a set of named pin the 'Value' pin of each counter bit, named 'VALUE' sub I

RELATIONS: {
 The logic equations FOR {L;R} SC CBITS; COLLECT
 { L\\$'RESET_IN' \EQU (R\\$'RESET_OUT');
 L\\$'FLIP_IN' \EQU (R\\$'FLIP_OUT') } }]

Connect the two adjacent counter bits named L & R

GUTS: CBITS END ENDEFN

Loop Generator: Set the variables L & R to each adjacent pair of counter bits from the array of counter bits CBITS.

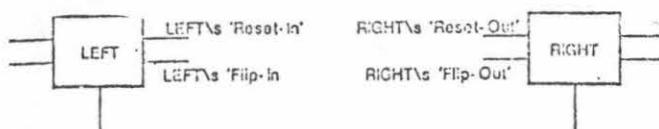
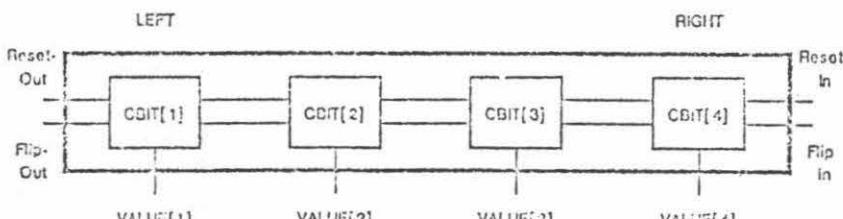


FIGURE 4

AND Terms

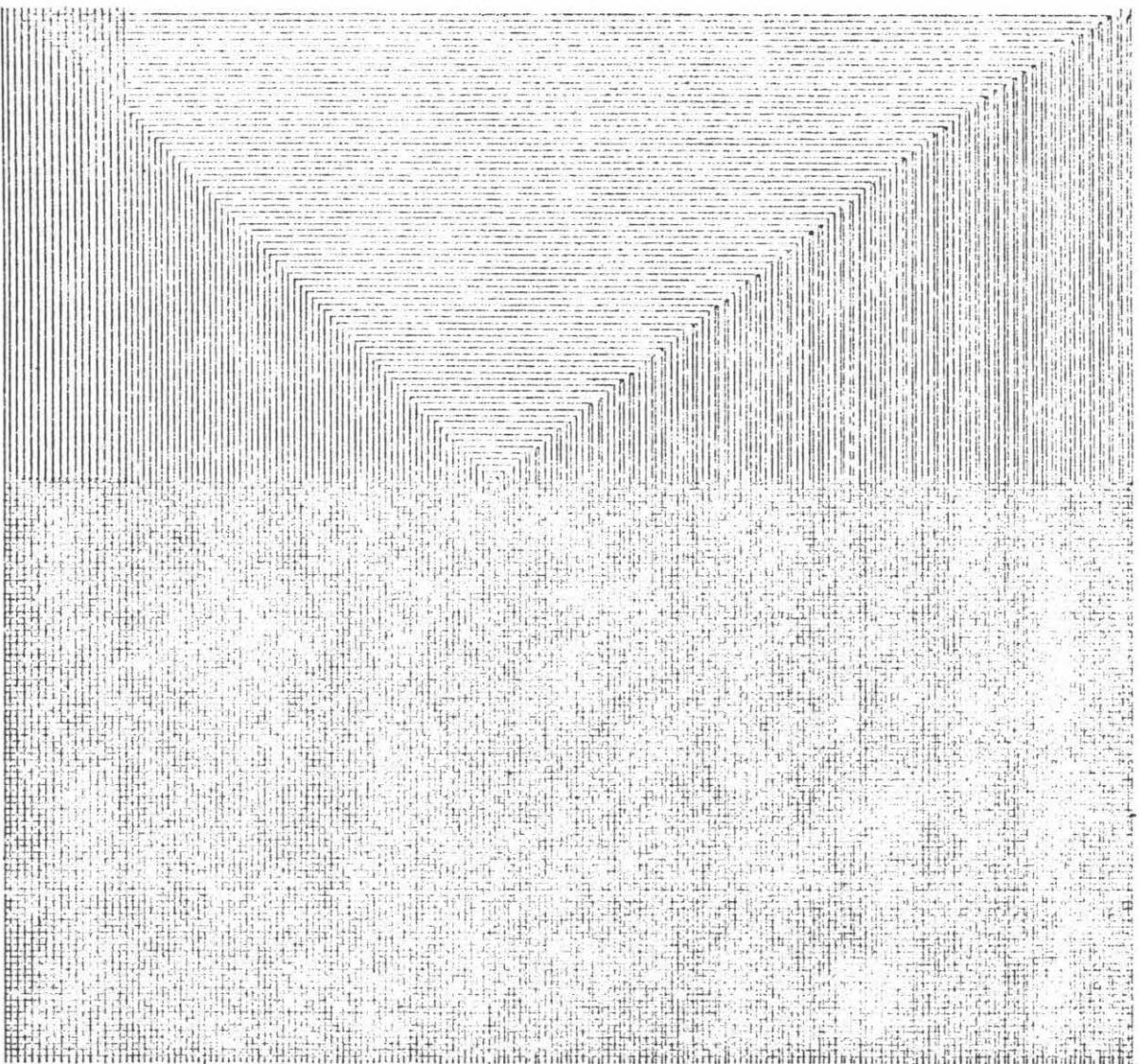


Figure 5

16-bit counter as
one PLA
area ≈ 32000

Figure 6
16-bit counter
area ≈ 1500

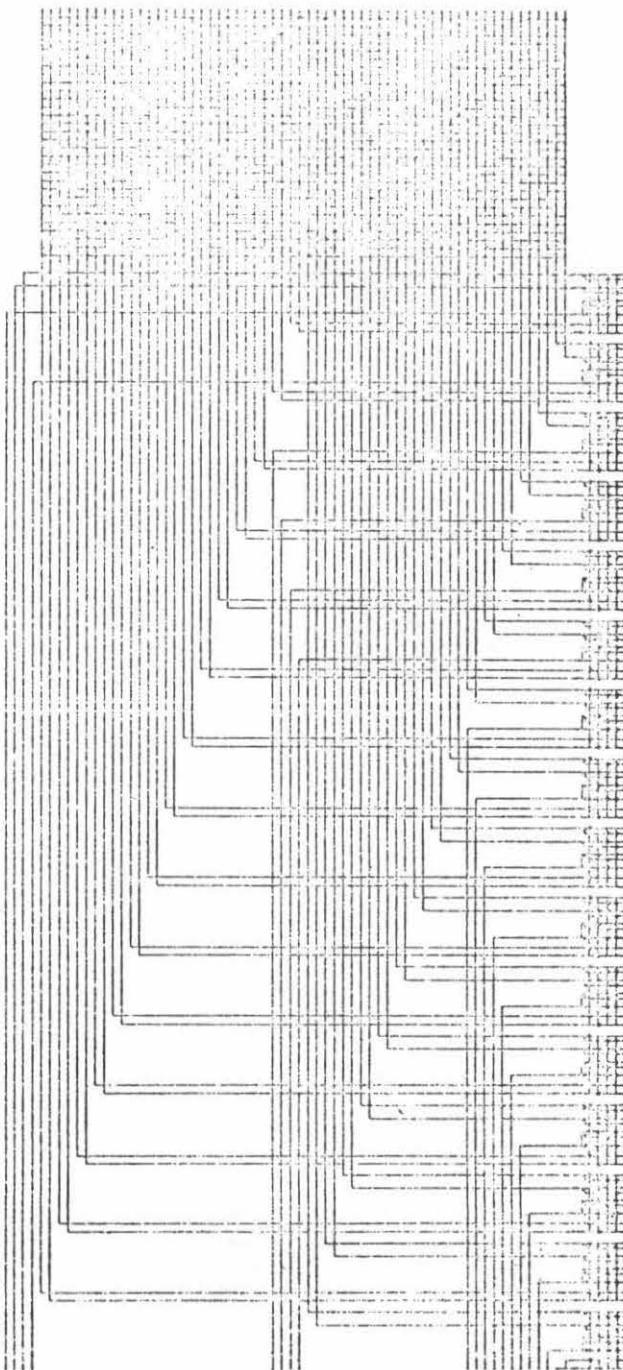


Figure 7

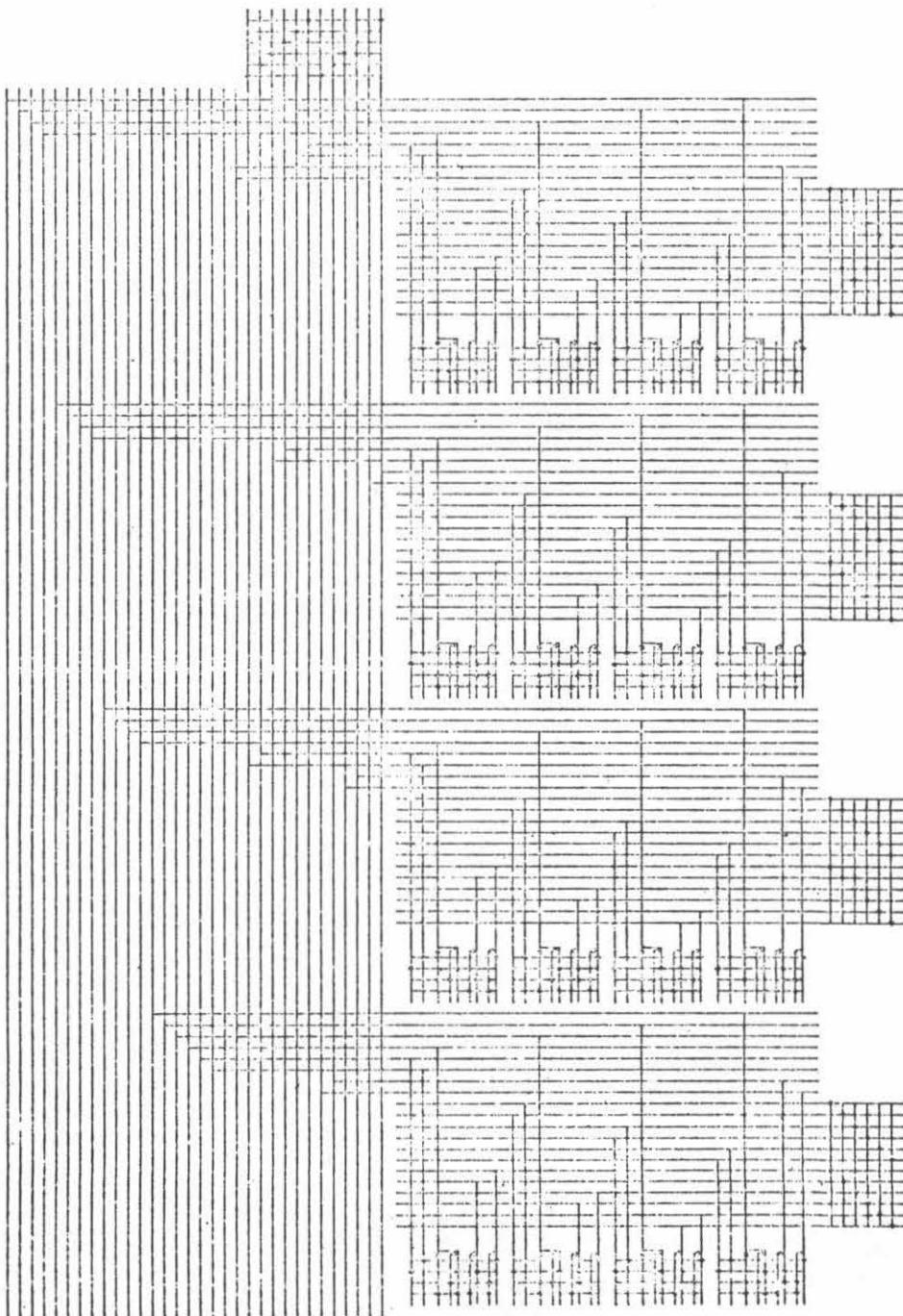


Figure 8
area ≈ 7000

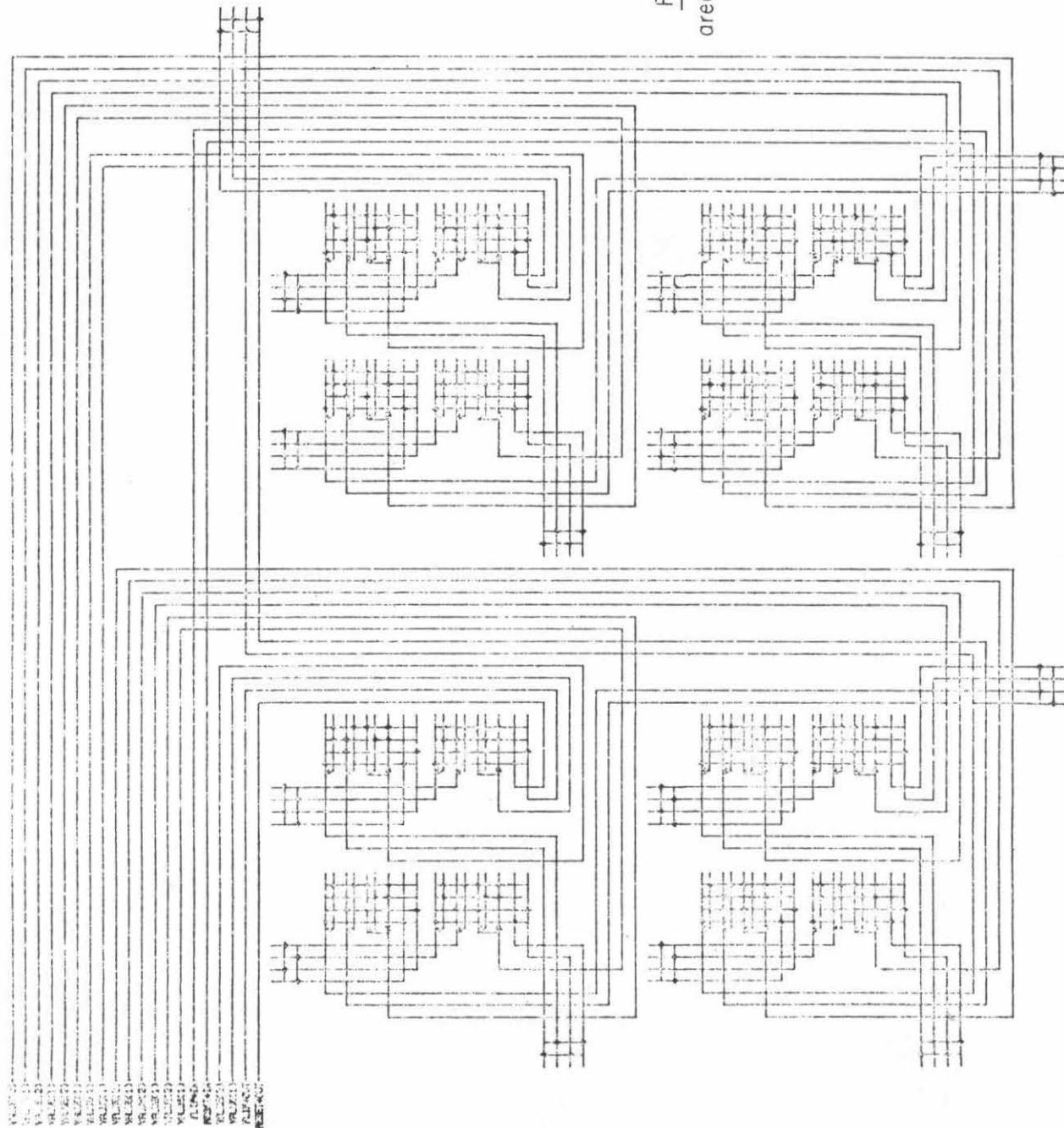


Figure 9
16-bit counter
with "trivial" equations removed
area ≈ 3000

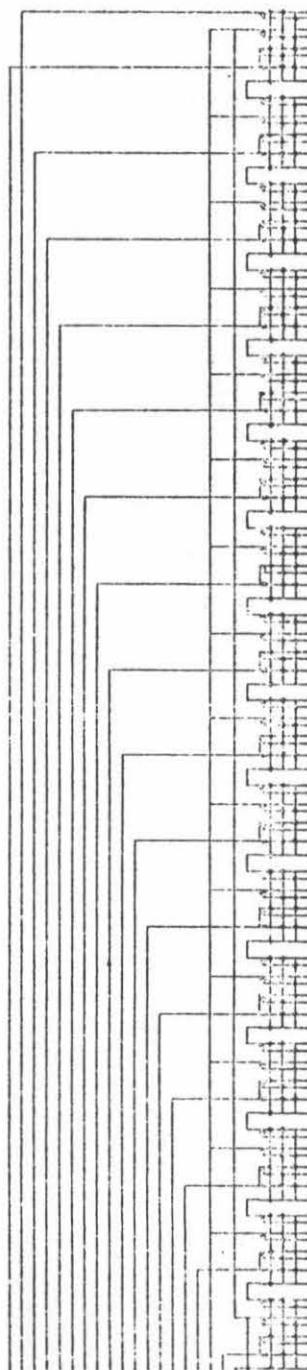


Figure 10
area ≈ 2200

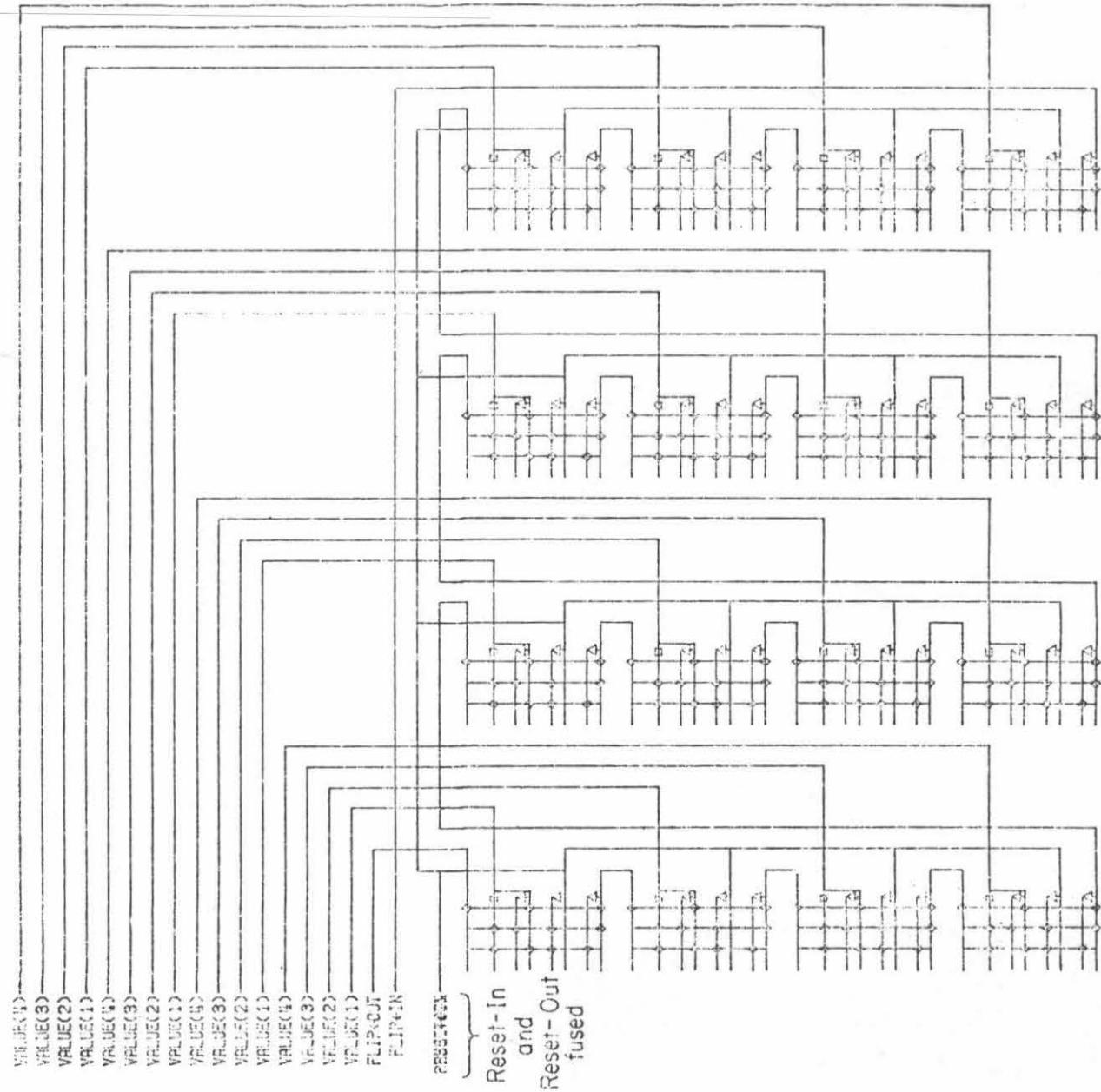
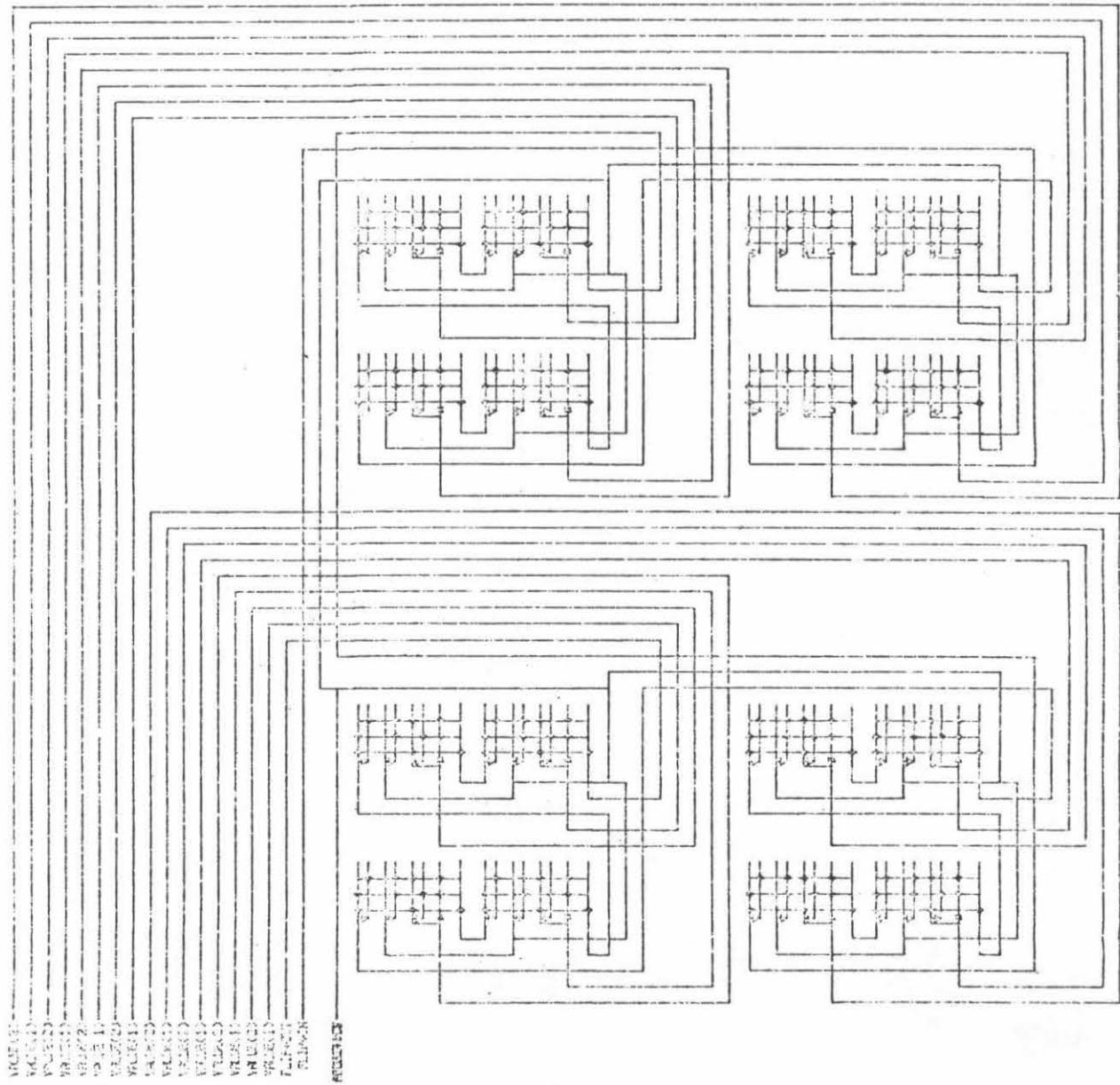


Figure II
area ≈ 4000



ADL : AN HIERARCHICAL LOGIC DESIGN LANGUAGE

Hilary J. Kahn, A. K. Burston and D. J. Kinniment

Department of Computer Science

University of Manchester

U.K.

1. INTRODUCTION

The use of Computer Aided Design techniques in the design of computer systems themselves is already well established, as can be observed from the widespread use of programs for the layout of PCBs and IC masks, automated production of design documentation and the use of automated manufacturing techniques [1]. Simulation, particularly digital component level simulation, has also proved an extremely useful tool in the development of computer systems [2]. The most successful of the CAD tools available have, inevitably, been at the lower (i.e. manufacturing) end of the design process where the problems are well understood and amenable to algorithmic solution. In the important area of test pattern generation and at the higher levels of the design process, the tools available have proved depressingly inadequate.

The Department of Computer Science, University of Manchester, has, for a number of years, specialised in the design of large, fast, computer systems, e.g. Atlas [3] and MUS [4], and has considerable experience of the practical difficulties in large system design, manufacture and maintenance. Following the successful use of CAD techniques in the design of MUS, a more comprehensive design system is currently under development. The main emphasis in the design of the CAD tools has been the provision of practical, usable (and hence used) tools rather than a more generalised system. It is felt that greater generality would require more time, manpower and computer resources than are readily available, and might still prove unsatisfactory in solving the real problems faced by the hardware engineer.

This CAD system is centred on a formal Data Base Management System, MUD [5,6] with non-programmer access via a flexible Command Processor [7]. The aim is to provide

- a high level hardware design language (ADL) translatable by machine into logic;
- a system level simulator associated with ADL;
- a lower level design language allowing 'hand-designed' logic to be incorporated;
- a gate level logic simulator which uses component models developed in a specialised logic description language;
- layout systems;
- documentation aids including a logic diagram package.

Part of the system is already in operation; much of the rest is currently under development.

1.1. Constraints on ADL

The main requirement of ADL (A Design Language!) is that it should be capable of describing large, fast, asynchronous systems in which a high degree of parallelism is inevitably present. At the same time, a hardware engineer using ADL would expect the system to automatically generate logic which used the highest speed technology readily available. The logic generated should, of course, be efficient both in time and volume. In order to help inter-designer communication, it is also intended that ADL should provide good design documentation.

ADL aims to provide a useful aid while still permitting the designer some freedom to develop new approaches when faced with new problems. Furthermore, the constructs of the language have deliberately been kept closely related to practical hardware implementations to enable the designer to have a 'feel' for the actual logic which will eventually be generated. This approach it is hoped will permit the skill of experienced designers to be used to the full.

1.2. Formal Design Methods

A number of high-level design systems [8] already exist. Some, such as ISP and PMS are more properly logic description systems and are too high level to be of practical use for automatic logic generation. Others, e.g. DDI, are aimed at synchronous or serial systems.

Although the two graphical approaches, Petri Nets [9] and LOGOS [10] are suited to parallel, asynchronous system design, they appear impractical for large systems. The pictorial approach makes it very difficult to examine more than a small part of the design at any one time. In addition, LOGOS, which uses two graphs - a data graph and a control graph - seems to need a significant amount of extra text to cross reference between the graphs.

2. THE STRUCTURE OF AN ADL DESIGN

A design expressed in ADL is hierarchical in that it is a block definition. Within that block definition reference may be made to constituent blocks (called subblocks) which may or may not already be in existence. Naturally, these subblocks can themselves be defined as ADL blocks with their own constituent subblocks. The lowest level of the hierarchy consists of basic subblocks such as registers, decoders, etc. Design of these low-level constituents is best done at the gate level rather than in ADL as they are conceptually simple and should be represented by the most efficient logic possible. An extendable library of these basic subblocks is held in the data base and includes many of the common primitives required.

Within an ADL block there may exist a number of control paths operating in serial or parallel as required. Use of appropriate branch and synchronisation constructs permits paths to diverge and converge. A given path is divided into alternate task and control sequence sections; a task specifies the set of concurrent events which are to occur when the task is active and a control sequence determines which task(s) require activation when the current task is deactivated. A task remains active until the control signal

combination for which the task waits has occurred. For example

```
T1 : 'FLOW' a ← b ,
      c ← d ;
      'SET' sig3, sig4;           Task T1
      'WAIT FOR' sig1 & sig2;
      → T5;                      Control sequence
```

Here, when T1 becomes active (as a result of a control transfer to it from some other task(s)), the data transfers from b to a and d to c are enabled and sig3 and sig4 set to '1'. This state persists until the control signals sig1 and sig2, are both set to '1' in response to the setting of sig3 and sig4. At this point, the task is deactivated and control is transferred unconditionally to T5.

3. ADL LANGUAGE CONSTRUCTS

The language features are summarised below; a more detailed description may be found in [11,12]. An example of the use of ADL is given in the appendix.

3.1. Static Declaration Section

An ADL block definition starts with a specification of the block interface and the interfaces of constituent subblocks. Note that a block interface is defined in terms of data ports and control signals. In addition, facilities exist to permit control signals local to the block to be defined as well as any invariant data paths. For example

```
'BLOCK' B ['INPUT' BIN [0:15]/BINREQ/BINACK/
            'OUTPUT' BOUT [0:7]
            'CONTROL IN' BA 'CONTROL OUT' BZ];
'SUBBLOCK' SB-
    SBOCC1 ['INPUT' SB1 [0:3], SB2 [-2:1] 'OUTPUT' SB0 [0:7]
            'CONTROL IN' SBCA 'CONTROL OUT' SBCX];
    'LOCAL CONTROL' LCA, LCB, LCC;
    'CONNECTION'     BOUT ← SB0;
```

This defines a block B which has a 16-bit input port, BIN, 8-bit output port, BOUT, and four control signals, BINREQ, BINACK, BA, BZ. Note that BINREQ/BINACK are specifically used to control data flow to port BIN and form a request/acknowledge pair which can be used to provide a 'handshake' signalling system between blocks. A subblock of type SB with two 4-bit input ports, an 8-bit output port and two control signals may or may not have already have been defined; an occurrence SBOCC1 is used with interface names SB1, SB2, SB0, SBCA and SBCX.

The 'local control' construct is used to define control signals additional to those defined as part of the interfaces of block B and subblock SBOCC1. LCA, LCB and LCC are available only within B and are useful in providing communication between various control paths and in controlling the internal timing of the block.

The 'connection' statement indicates that data ports SBO and BOUT are to be permanently interconnected and hence no further transfer-enabling logic will be required. In its most general form, this construct can be used to define quite complex data port connections.

Any digital system must be set to a predefined state when initially 'powered up' in order to ensure correct operation. In ADL, it is assumed that a 'general reset' signal will exist and that all tasks and control signals will be made inactive unless specifically excluded by use of an INITIALIZE or INITIALIZE CONTROL statement.

3.2. Control Section

This section contains task definitions and control sequences.

3.2.1. Tasks

Tasks are delimited by a task label and a timing statement such as WAIT FOR

e.g. T1 : statements
 'WAIT FOR' LCA + LCB;

The effect of the WAIT FOR is to suspend control within T1 until the appropriate signal state combination has occurred. Note that all control signals in ADL have an associated flag and that when the task is deactivated, once either LCA=1 or LCB=1, the flag for the relevant control signal is reset.

The most common statements which occur within a task are FLOW, SET and RESET which allow data paths and control signals to be modified. For example

- (i) 'FLOW' SB1 ← BIN [12:15];
causes a temporary interconnection between a part of BIN and SB1. This interconnection is only enabled when the task in which the statement appears is active. Note that a single port may have data 'flowed' to it from many different sources at different times and logical operators may be used to combine data ports.
- (ii) 'SET' LCA; or 'RESET' SBC1, LCB;
These permit explicit setting/resetting of control signals in order to enable communication between separate tasks.

3.2.2. Control Transfer

A range of control transfer instructions is available to permit control paths to branch conditionally or unconditionally. The most basic of these is → . For example

→ T4 transfers control to task T4
→ (T3, T9) transfers control to T3 and T9 simultaneously.

The control transfer may be made conditional by prefacing the '→' with 'IF' condition 'THEN'. For example

```
'IF' SBO = 0 'THEN' → T8;
```

A 'no destination' statement, *, is available to terminate a control path.

A more complex conditional, DECODE, provides a parallel control path switch based on the state of a data port. For example

```
'DECODE' BIN [3:6] → [T3, T5, T6, (T7, T10)];
```

transfers control to all destinations for which the corresponding bit of BIN is at a logical 1.

The basic '→' statement and the simple conditional version may be made to operate in either parallel (//) or serial (#) mode. For example

```
T1 : statements
      'WAIT FOR' condition 1;
      // → T4;
      # 'IF' condition 2 'THEN' → T3;
          → T2
```

Here T4 will always be activated; T3 will be activated if condition 2 is true otherwise control is transferred to T2.

The constructs discussed so far provide most of the facilities needed by the hardware designer. However, there remain two problem areas of considerable importance to the designer of complex, parallel systems; priority resolution and the control of mutually exclusive access to a shared resource.

3.2.3. Priority Handling

It is typical of parallel systems that a control path may be activated from a number of different positions. Assuming only one activation at a time is permitted, a decision must be made about which activation to allow. In ADL this is done by inserting a special priority mechanism in the control path. This mechanism consists of a PRIORITY WAIT which appears inside a task (instead of a WAIT FOR) and makes use of a PRIORITY BLOCK to do the decision making. For example

```
'PRIORITY BLOCK' PB-
    PBOCC [3];
```

can be used to decide between three conflicting requests and might be accessed by

```
T5 : 'PRIORITY WAIT' PBOCC-
      P1 : 'WHEN' LCA 'THEN' → T6,
      P2 : 'WHEN' LCB 'THEN' → T7,
      P3 : 'WHEN' LCC 'THEN' → T8;
```

In this example, suppose T5 is active when one or more of the control signals occurs. The states of all three control signals are staticised and input to PBOCC together with a special 'make a decision' signal. After a delay, a 'decision made' signal will be generated and a wire corresponding to an active control signal will be set so that control can be passed to one of T6, T7 or T8. If required, data ports can be used by the priority block to alter

the decision making criteria.

3.2.4. Mutual Exclusion

This problem is one of preventing simultaneous access to a shared resource by two or more parallel control paths. The ADL solution to this problem uses the hardware equivalent of a semaphore. A special controlled priority block which is used by two or more priority waits must be defined. The priority waits are at the start of the sections of control path concerned with accessing the shared resource. The sections are called the 'critical sections' of the relevant paths. For example

```
'CONTROLLED PRIORITY BLOCK' CPB-
CPBOCC [2];

T1 : 'PRIORITY WAIT' CPBOCC-
P1 : 'WHEN' LCA 'THEN' → T2;
T41: 'PRIORITY WAIT' CPBOCC-
P1 : 'WHEN' LCB 'THEN' → T42;
```

Assuming both T1 and T41 are active, when either LCA or LCB is set a priority decision is made and control continues with T2 or T42. The other path is suspended and CPBOCC is 'locked'. When the active path no longer requires the common resource, it issues a release statement such as 'RELEASE' CPBOCC. Any outstanding requests to the priority block will then be considered.

4. IMPLEMENTATION

4.1. The ADL Translator

A logic design expressed in ADL is input to the translator which applies syntactic checks and produces as output an intermediate data structure (IDS) which can be used by a number of different programs including the logic generator. The IDS, which is stored in the data base, is a set of tables which are closely correlated with the original text except that certain duplication is avoided. For example, if the same flow statement appears in more than one task the flow information is stored once only so that the logic generator need only create one version of the logic.

4.2. The ADL Logic Generator

The logic generator uses the IDS to create the logic for an ADL block. It operates in two passes and requires that the interface details of the block and of any constituent subblocks be fully defined. These details include loading and fan-out constraints which can be input via the Command Processor.

During the first pass, the IDS is examined and idealised 'meta-logic' is produced. The main logic synthesis is carried out at this stage but practical constraints of fan-in and fan-out are ignored. The second pass of the logic generator transforms the meta-logic into the particular technology required and adjusts the gating to take account of fan-in, fan-out, inversion and loading. This approach, which is commonly used to aid portability in programming languages, is flexible and localises the effects of having to generate logic using a number of rapidly developing technologies. An example of part of the logic generated for the design given in the Appendix is shown in Fig.A1.

It should be noted that the ADL logic generator does not need to fill in the detailed logic for subblocks within the block being examined. A separate integrating program assembles a complete network from the logic information which is stored in the data base for each of the blocks and subblocks.

4.3. Meta-logic

ADL language constructs are represented by combinations of simple function modules which are, in principle, implementable in any technology. The full set of these meta-logic modules and some typical implementations are shown in Fig.1. The purpose of each module is summarised as follows :

Task	Initiates the functions (e.g. FLOWS, SETs) within a task and waits for task completion.
Signal	Provides a static flag to indicate the occurrence of a control signal.
If	Propagates one of two control paths depending on a data condition.
Decode	A group of decode modules selects a subset of control paths to propagate.
Edge buffer	Converts an edge to a level for testing.
Flip-flop and Delay	Used to staticise signals to be tested inside priority wait statements.

In addition, three basic gate types, AND, OR and EQUIVALENCE, which are assumed to have infinite fan-in, are available. Further details of the operation of the modules may be found in [7].

5. CONCLUSIONS

A number of experimental logic designs have been developed to test the system. Although the quality and efficiency of the generated logic is yet to be evaluated, experience indicates that ADL is convenient to use and provides a viable formalism for expressing the concepts of logic design.

Future work planned includes implementation of the logic integrating program, production of a system level simulator and an automatic diagram drawing package to provide graphical output to accompany an ADL design.

REFERENCES

- [1] de Man H. :
"Computer Aided Design : Trying to Bridge the Gap"
European Solid State Circuits Conference, Amsterdam, 1978.
- [2] Kahn H.J. and May J.W.R. :
"The Use of Logic Simulation in the Design of a Large Computer System"
Radio Electron. Eng., Vol.1, 497-503, 1973.
- [3] Lavington S.H. :
"The Manchester Mark I and ATLAS - A historical perspective"
CACM, Vol.21, No.1, 1978.
- [4] Ibbett R.N. and Capon P.C. :
"The Development of the MUS Computer System"
CACM, Vol.21, No.1, 1978.
- [5] Wilson T.B. :
"A Data Description Language"
M.Sc. Thesis, University of Manchester, 1974.
- [6] Wilson T.B. :
"A Data Base Management System for the MUS Computer"
Ph.D. Thesis, University of Manchester, 1976.
- [7] Burston A.K. :
"An Integrated Logic Design System"
Ph.D. Thesis, University of Manchester (to be submitted).
- [8] Special Issue on Hardware Description Languages
COMPUTER, December, 1974.
- [9] Petri C.A.:
"Kommunikation mit Automaten"
Schriften des Rheinsch - West-Fälischen Inst. für Instrumentelle
Math., Univ. Bonn, 1962.
- [10] Rose C.W., Bradshaw F.T., Katze S.W. :
"The LOGOS Representation System"
IEEE Proc. COMPCON, September 1972.
- [11] Burston A.K. :
"The Development of a Computer Logic Design Language"
M.Sc. Thesis, University of Manchester, 1975.
- [12] Burston A.K., Kinniment D.J., Kahn H.J. :
"A Design Language for Asynchronous Logic"
Computer Journal, November 1978.

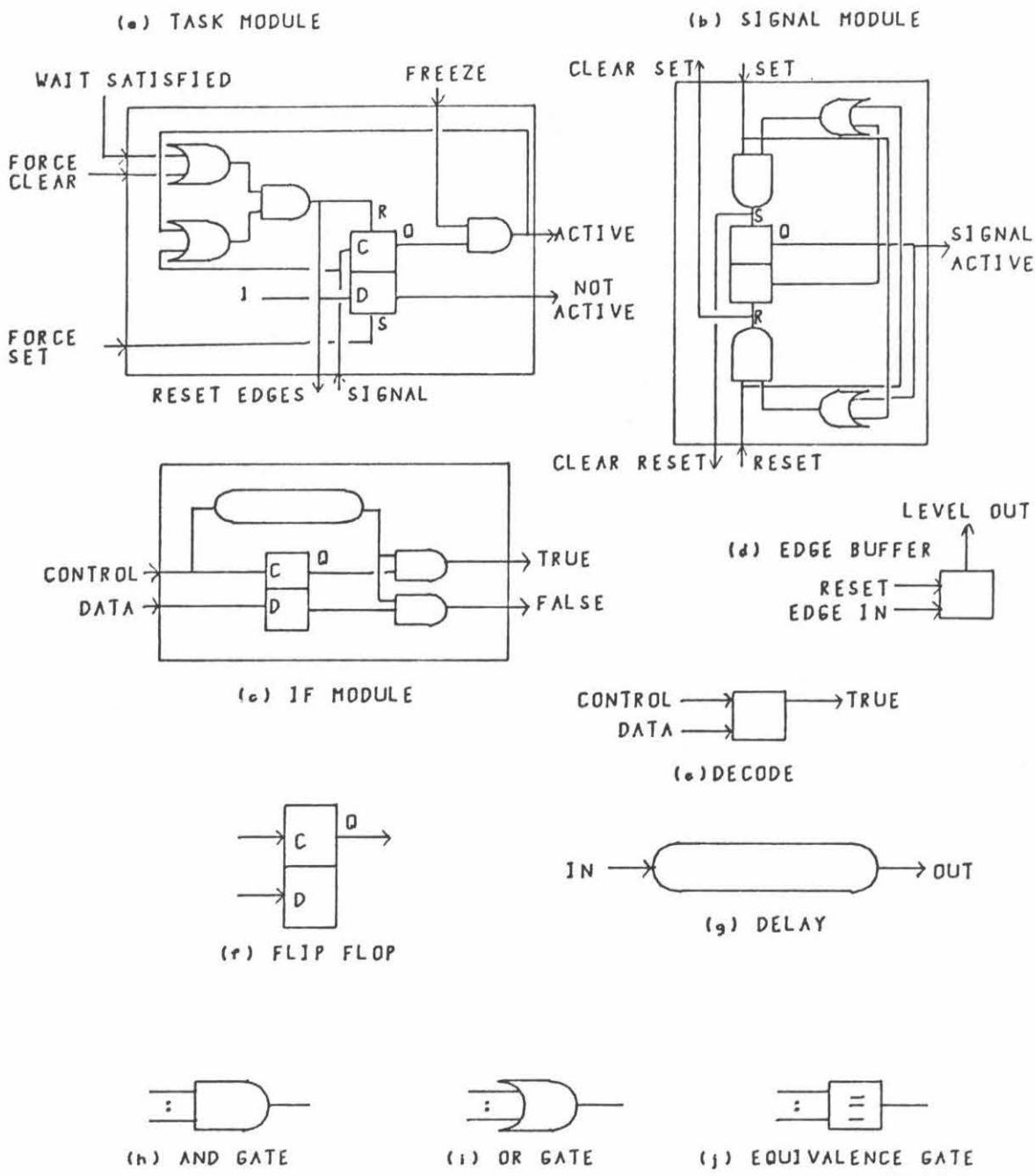


Figure 1 : Meta-logic Modules

Appendix - An Example of ADL

The example is the design of a subblock to perform the "compression" function. The operation is defined as follows: given two equal length bit vectors MASK and DATA a vector of less than or equal length, RESULT, is produced from DATA by suppressing all the bits of DATA for which a "0" appears in the corresponding position in MASK. For example:

DATA	10110101
MASK	01100110
RESULT	01 10 = 0110

The subblock works on 8 bit elements, unused bits of RESULT are zero filled. The output of the unit is buffered, that is, a second calculation can be performed whilst the result of the first is held on the output, ready to be accepted by the outer block. Handshake control is used throughout.

The overall operation of the unit consists of three parallel control paths. The first is the main loop (T1, T2, T4, T5, T6, T7, T8) which is concerned with shifting the data and setting RESULT. The second (T3) is concerned with counting the number of iterations performed. The third (T9) is concerned with buffering the output.

```
'BLOCK' COMPRESS['INPUT' MASKIN[0:7],DATAIN[0:7] 'OUTPUT' DATAOUT[0:7]
    'CONTROL IN' GO, ACCEPTED 'CONTROL OUT' TAKEN, DONE];
!MASKIN - 8 bit input port for the MASK.
DATAIN - 8 bit input port for the DATA.
DATAOUT - 8 bit output port for the RESULT.
GO - start calculation, input data ready.
TAKEN - calculation performed, ready for new input data.
DONE - output data ready for taking.
ACCEPTED - output data taken, may now change;

'BASIC SUBBLOCK' EXREG -
A['INPUT' AIN[0:7] 'OUTPUT' AOUT[0:7]
    'CONTROL IN' LOADA,SHIFTA 'CONTROL OUT' LOADADN,SHIFTADN],
B['INPUT' BIN[0:7] 'OUTPUT' BOUT[0:7]
    'CONTROL IN' LOADB,SHIFTB 'CONTROL OUT' LOADBDN,SHIFTBDN],
C['INPUT' CIN[0:7] 'OUTPUT' COUT[0:7]
    'CONTROL IN' LOADC,SHIFTC 'CONTROL OUT' LOADCDN,SHIFTCDN],
D['INPUT' DIN[0:7] 'OUTPUT' DOUT[0:7]
    'CONTROL IN' LOADD,SHIFTD 'CONTROL OUT' LOADDDN,SHIFTDDN];
!EXREG - general purpose shift register with parallel load.
IN - 8 bit parallel input port. OUT - 8 bit parallel output port.
LOAD - start load cycle. LOADDN - load cycle complete.
SHIFT - start shift, output is shifted one place left, top bit is lost,
        bottom bit is replaced by bit on bottom end of IN.
SHIFTDN - shift cycle completed.
registers: A - MASK, B - DATA, C - RESULT, D - output buffer;
```

```

'BASIC SUBBLOCK' COUNTER - 

COUNT7['OUTPUT' ZERO 'CONTROL IN' SET7,DEC 'CONTROL OUT' SET7DN,DECDN];

!COUNTER - down counter.
ZERO - equals "1" if counter contents are zero.
SET7 - set counter contents to 7. SET7DN - contents reduced by one.
DEC - reduce contents by one. DECDN - contents reduced by one;
'LOCAL CONTROL' SUBTRACT, AVAIL;
!SUBTRACT - set when a decrement of the counter is complete.
AVAIL - set when the output buffer is empty;

'CONNECTION' AIN <- MASKIN, BIN <- DATAIN, DIN <- COUT, DATAOUT <- DOUT;

'INITIALIZE' T1; 'INITIALIZE CONTROL' AVAIL;

'BEGIN';

'DECISIONS'; !decide if the current bit is to be saved;
D1: 'IF' AOUT[7]=0 'THEN'-> T5;
    -> T4;
'END DECISIONS';

T1: !wait for input, indicate ready;
    'SET' TAKEN;
    'WAIT FOR' GO;

T2: !initialize counter, RESULT, get input;
    'FLOW' CIN <- @00; 'SET' LOADA,LOADB,LOADC,SET7;
    'WAIT FOR' LOADADN&LOADBDN&LOADCDN&SET7DN;
    -> (T3,D1);

T3: !decrement counter;
    'SET' DEC;
    'WAIT FOR' DECDN;
    'SET' SUBTRACT;
    *; !terminate this control path;

T4: !transfer one bit from DATA to RESULT;
    'FLOW' CIN[0]<-BOUT[7]; 'SET' SHIFTC;
    'WAIT FOR' SHIFTCDN;

T5: !wait for end of cycle;
    'WAIT FOR' SUBTRACT;
    'IF' ZERO 'THEN' -> T7;
// -> T3;

T6: !shift DATA and MASK up one place;
    'SET' SHIFTA, SHIFTB;
    'WAIT FOR' SHIFTADN&SHIFTBDN;
    ->D1;

```

```
T7: !wait for output buffer to become free;  
    'WAIT FOR' AVAIL;  
  
T8: !transfer RESULT to output buffer;  
    'SET' LOADD;  
    'WAIT FOR' LOADDON;  
    -> (T1,T9);  
  
T9: !indicate output available and wait for reply;  
    'SET' DONE;  
    'WAIT FOR' ACCEPTED;  
    'SET' AVAIL;  
    *;  
  
'END';
```

Figure A.1 shows the generated logic for the section of control surrounding task T5.

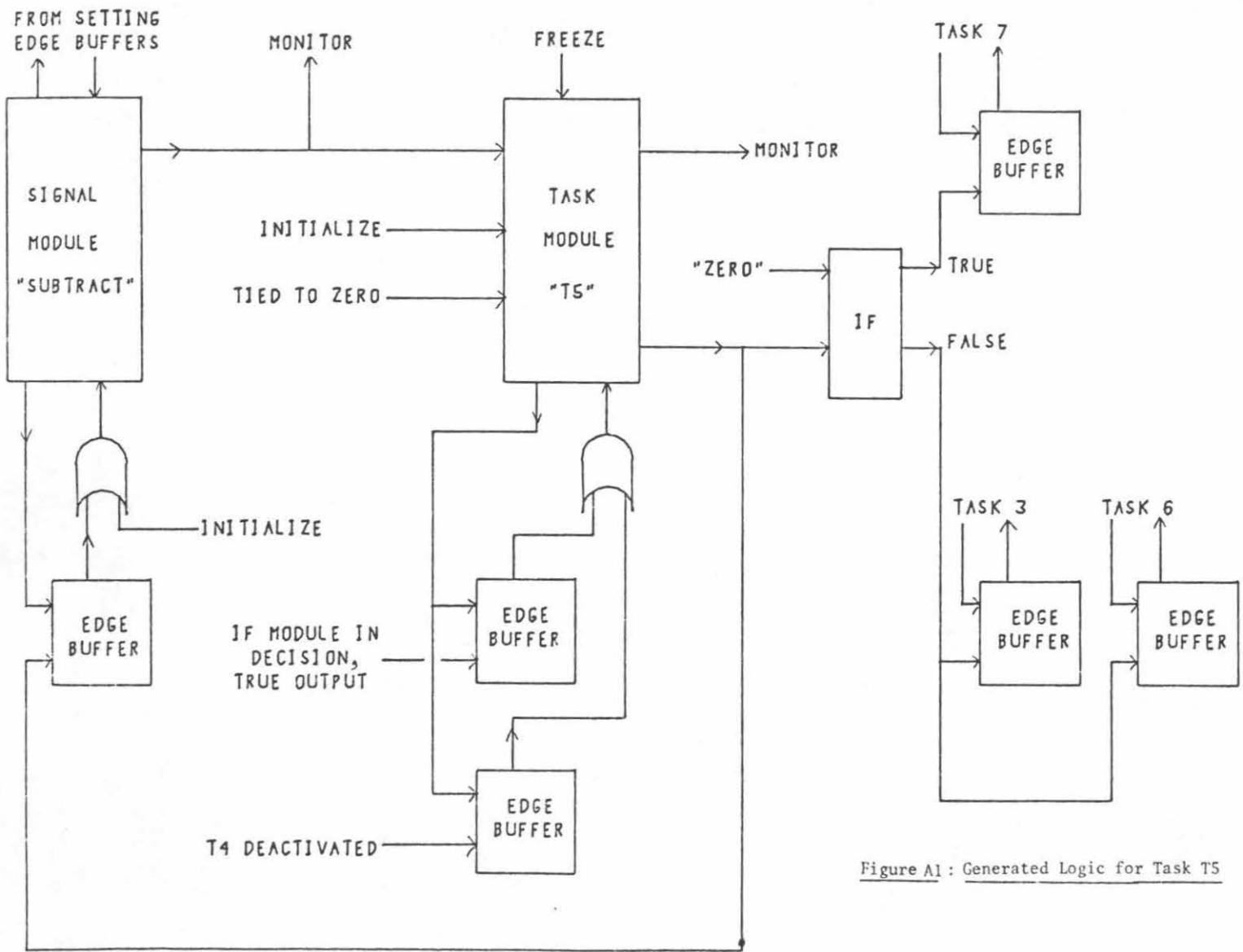


Figure A1 : Generated Logic for Task T5

SELF-TIMED LOGIC SESSION

Chairperson:

Charles E. Molnar, Professor of Physiology, Biophysics, and Computer Engineering, Director of the Computer Systems Laboratory, Washington University; and Visiting Professor of Computer Science at Caltech

The appearance of this session on Self-Timed Logic in a Conference on Very Large Scale Integrated System Design may warrant some explanation. The effects of the downward scaling of dimension, and the corresponding changes in the relationships between propagation times through logic circuits and along interconnecting pathways, require a careful re-examination of the means chosen for the control of sequencing of operations within single-chip and multi-chip systems.

At the highest levels of system design, it is convenient to suppress implementation detail through the use of design and specification models that are topological in nature, and express connectedness and sequencing in the form of topological models that do not contain any metric information regarding space or time. Nevertheless, at some point in the progress of design toward the ultimate fabrication of physical systems implemented in silicon, it is necessary to recognize that the resulting structure must satisfy partial differential equations in space and time.

The relating of topological models to physical implementations has often been done most conveniently through the imposition of a regular and often periodic subdivision of the space and time continua. In the spatial setting, this approach has been seen in papers in the previous sessions that have described regular cellular structures, such as the master slice and polycell approaches. In the temporal setting, the association of occurrences of system events with particular instances of regular system-wide clocking signals is a standard and often convenient means for relating ordering specifications for the steps in a process to the passage of time.

The first paper of this session, by Seitz, gives us some perspectives on the effects of increases of circuit density upon the numbers of elements in a system and upon the relationships between the propagation times through logic circuits and along interconnecting pathways. He poses some important questions about the practicality of system-wide clocking in VLSI systems.

The next paper, by Chaney and Rosenberger, points out some hazards that arise in the interconnection of sub-systems that do not share a common clock, and presents some methods and results that can be used to estimate the severity of these hazards to reliable system operation.

The following Stucki and Cox paper clarifies the choices that must be considered by the designer who wishes to follow a rational strategy for synchronization, and points out their implications for system reliability and performance.

The fourth paper by Sutherland, Molnar, Sproull, and Mudge presents the design of a bus structure that attempts to match the circuit properties of highly integrated MOS interconnections so as to obtain an efficient and well-specified intercommunication structure that is entirely self-timed.

The final paper, by Patil, presents a systematic and general approach to the direct realization of self-timed control circuits from Petri Net descriptions, using a PLA-like structure that is well suited to integrated circuit implementation.

Taken together, these papers provide only the slightest introduction to the topic of self-timed logic in VLSI systems. Whether the regularity of the currently more popular and more highly developed synchronous approaches to system timing will prove to be an optimum way of controlling complexity and reducing design detail in the VLSI era, or will instead appear in retrospect to be the artificial imposition of an irrelevant and distracting temporal texture whose major virtue is allowing the use of integer arithmetic in our computer design aids is still a matter for debate.

SELF-TIMED VLSI SYSTEMS

Charles L. Seitz

Department of Computer Science
California Institute of Technology*Abstract & Introduction*

This short paper is intended to explain why the subject of self-timed logic is relevant to a conference on VLSI.

Scaling down feature size and scaling up chip area not only increases the complexity of chips, but also changes relationships in the parameters which describe the physical characteristics of switching devices, circuits, and wires. The physical change which most impacts the design disciplines employed for VLSI -- particularly the timing aspect of design -- is the increased wire delay associated with the increased resistivity of scaled down wires. Wires that run even a small fraction of the way across a chip will impose a significant delay. Clock distribution and long-distance communication required by synchronous systems will become problematic. Otherwise, it appears that the timing aspect of design for submicron feature size circuits will generally resemble that of today's MOS technology, in that delays will be largely determined by parasitic wiring capacitance.

A general introduction and outline of the self-timed discipline of digital system design (by this author) is to be published in Chapter 7 of [1], and to which the reader is referred for more detail. Briefly, the self-timed discipline is concerned with complexity management, and with timing of digital systems under conditions in which delays between parts are large or uncertain. The amount of circuitry that can be fabricated onto a single chip is increasing to well beyond that point at which informal complexity management disciplines are effective. The escalation of complexity suggests for VLSI design a rigorous discipline of modularity. The large long-distance communication delays suggest in addition that the modules be independently timed. This is the central idea of self-timed systems: it is a discipline in which each system "part" keeps time to itself. Those "parts" of a self-timed system whose correct operation must be certified by appealing to time and physical argument are called *elements*. Otherwise, self-timed systems are constructed (defined recursively) as restricted classes of interconnections of self-timed systems or elements, and correct *sequential* operation is independent of the delays in the interconnections.

Scaling Microcircuit Timing

A basic reference on the physical consequences of scaling feature size is *Introduction to VLSI Systems* [1] by Carver Mead and Lynn Conway, or the article [2] by Amr Mohsen in the first section of these *Proceedings*.

What is meant here by digital system "timing" is the way in which the

This research was supported by the Defense Advanced Research Projects Agency (ARPA) under contract number N00123-78-C-0806.

physical, metric notion of time is connected to a sequential process. The correct operation and performance of a switching system depends upon the values of and tolerances on:

- (1) the switching delay of the elementary switching devices,
- (2) the additional delay induced on the switching devices by attaching wires to them and incorporating them into circuits, and
- (3) the delay associated with equalizing the potential across wires.

Scaling feature size shifts many of the physical parameters which determine these delays. The scaling effects discussed in some detail below are those that appear to have a major impact on timing and design disciplines, and are too fundamental to be circumvented entirely by process "fixes."

MOS Switches and Circuits

As indicated in the references [1,2] cited above, sensible scaling of MOS technology holds electric fields constant, so that power and signal voltages scale down with the minimum feature size. The lithography directly determines the minimum gate length of the MOS transistor, so the transit time of this switch and currents also scale down in proportion to the feature size. Even though smaller MOS devices are proportionately faster, power per device scales down quadratically, and power per unit area remains constant. These general scaling rules apply to MOS parts in which performance is an issue. Of course, some applications trade speed and power in favor of reduced power consumption, and so operate at lower voltages.

Switching energy, the product of the device power and its delay time, is a fundamental figure of merit which relates the cost of a computation to this parameter of the switching devices which perform the computation. The switching energy for MOS technologies scales down as the third power of the feature size, so the general physical consequence of reduced feature size is highly desirable. The only unfavorable effect of this scaling for the switches relates to the reduced operating voltages decreasing gain and increasing subthreshold current. These characteristics are exponentially dependent on the ratio of signal voltages to kT/q , and so could be brought into line by operating circuits at reduced temperature. Dynamic storage requires small subthreshold currents and presently places the upper bound on many timing relations. Although dynamic registers will continue to be an important MOS circuit trick, scaling will reduce the refresh period to the extent that the dynamic random-access storage devices used so widely today will become unworkable and replaced with static designs.

What is generally left out of such discussions of the consequences of scaling on the *switches* is the following important characteristic of MOS technology: If an entire MOS circuit or system is scaled down -- switches, wires, voltages, and all -- its timing behavior scales essentially with that of the switches. We mention as exceptions only that the scaling must be such that wires can still reasonably be regarded as equipotentials and increased subthreshold currents do not discharge dynamically stored information.

Consider the process of scaling a MOS circuit such as an adder or PLA. Assume that the physical structure is shrunk in all three dimensions, not

just in the plan view of the circuit, but in the thickness of the various conductive and insulative layers as well. This form of scaling maintains the same relative surface flatness for the lithographic process, and preserves the electric field geometries of the original circuit. When one of the wires internal to the circuit is shrunk, its (charge-carrying surface) area scales down quadratically while its separation from substrate and from other conductors scales down linearly. So, the capacitance of the scaled down wire scales down linearly. Of course, so too does the gate capacitance scale down linearly. Please note for future purposes that the capacitance per unit length of a minimum dimension wire remains constant in the scaling.

Now, if each wire in the scaled down circuit has its capacitance scaled down linearly, and the current from the switches also scales down linearly, the time rate of change of voltages should remain reasonably constant in the scaling. However, the voltages, hence the voltage difference between logic zero and one, also scales down linearly, so the time required to accomplish a voltage transition between logic zero and one scales down linearly with the reduced feature size and with the transit time of the MOS switches.

Accordingly, smaller MOS circuits and systems are also proportionately faster temporal replicas of today's designs, just as the switches are, but for the possible exceptions mentioned above.

Bipolar Switches and Circuits

The bipolar transistor is normally built up vertically through a sequence of diffusions. The area of the base of each transistor is defined by the lithography, but the thickness of the base is determined by the difference between diffusions, and has for years been about as thin as possible. Fixed base thickness implies that voltages in scaled bipolar circuits could remain the same as those presently in use (but it appears that only those circuit families which use small signal voltage changes will be used in VLSI designs). The bipolar junction produces a maximum "on" current density perpendicular to the surface of the chip, so a reduction in minimum feature size produces a quadratic reduction in base area and in current for the minimum dimension transistor. So again, this scaling produces a quadratic reduction with feature size in the power per device, and constant power per unit area. However, the switches are no faster, and the switching energy improves only quadratically.

The current available from bipolar switches is today in a favorable relationship with the wiring capacitance, but this situation worsens as feature size is reduced. Since the currents scale down quadratically in any scaled circuit, while voltage changes remain fixed, wiring capacitance would have to drop quadratically with feature size just to maintain the same performance. However, since one cannot expect the capacitance of an internal circuit wire to scale down faster than linearly as the circuit is shrunk, scaled down bipolar circuits and systems can be expected to become slightly slower with smaller feature size. Of course, the increased significance of parasitics for bipolar circuits with smaller feature size may be offset by improved layouts or processes.

Signal Energetics

While it is usually defined as the product of the device power and delay time, the switching energy is also proportional to the energy that is required to change the state of a switch input. Reducing the switching energy generally reduces the cost of performing a computation, in that it reflects a reduction in the resources (energy, area) or time required. The reduction of feature size is a very direct approach to reducing the cost of a computation by reducing the switching energy, but equally significant is the reduction in wire size and parasitic energy.

The difference in significance of scaling parasitic delays in MOS and bipolar technologies can also be deduced from an energetics viewpoint. When voltage, current, and transit time all scale down linearly with the minimum feature size dimension as they do in the form of MOS scaling discussed here, the switching energy then scales down as the third power of the feature size. One can reach the same conclusion from the energy stored on the gate of a MOS transistor varying as its capacitance, which scales down linearly, times the square of the voltage. The energy associated with the parasitic capacitance of wires scales in exactly the same way -- indeed, the gate is part of a wire --, which accounts for the temporal behavior of a MOS circuit scaling exactly with that of the switches. What happens in the scaling of bipolar technology is that the switching energy scales down quadratically, as was described in the previous section, and which can be seen also from the capacitance of the bipolar transistor base scaling down quadratically with its area while the voltages remain fixed. The relatively increased significance of parasitic capacitances and energies is that they scale down only linearly.

Whether of MOS or bipolar technology, the same scaling down of the physical dimensions and switching energies that reduces the cost of performing a computation also creates a large disparity between the internal signal energies and those required at package pins. The driver circuits which must be used to increase signal energy levels at the package pins introduce an exceptional cost in the communication between chips, both in area and in delay (not to mention packages, lead-bonding, etc.). Given a particular transistor characteristic, the minimum driver delay varies as the (natural) log of the ratio of output and input signal energies [1]. This ratio is today typically 100 to 1000, for which the log is in the range 4.5 to 7 (representing the optimum number of stages in the driver). Another factor of 10 to 100 in this ratio, the expected consequence of reducing internal signal energies, will not make a dramatic difference in optimum driver delay, because of the logarithmic dependence. This relationship may explain why little attention has been given to chip packaging schemes which reduce interchip interconnect capacitance. Today's pin driver circuits are a compromise between area and delay, and it is not clear whether delay or area will be more precious in the future.

The pin-driving problem scales somewhat differently for MOS and bipolar technologies. In MOS technology, in which one expects the signal voltage changes at package pins to scale down as they do internally, the energy required to switch the binary voltage on a package pin of given capacitance scales down quadratically with feature size. Thus the ratio of package pin to internal signal energies scales up linearly, and one would expect a

modest increase in the delay imposed by an optimum driver, relative to transit time. However, MOS transit times decrease in the scaling, so the absolute delay of a pin driver circuit can be expected to decrease, although not quite with the transit time. In bipolar technology, in which one expects the signal voltage changes cannot be reduced to less than the several tenths of a volt changes now used in those circuit families showing smallest switching energies, the package pin signal energies remain constant while the internal signal energies scale down quadratically, or perhaps only linearly when dominated by the energy required to drive parasitic wiring. Thus the impact of scaling on pin-driver delay is slight in both the relative and absolute senses.

The Submicron Technology

Although MOS technology benefits more from scaling down feature size than does bipolar technology, it should be understood that MOS started its development from an inferior position in the physical measures of its switching devices. At minimum feature size much above one micron, the two technologies show very different strong points, which have helped to distinguish their areas of principal application. Bipolar technology exhibits superior performance, measured either in speed or in switching energy, and relative insensitivity of circuit operation to wiring capacitance and layout. Some design strategies that today work well for bipolar and poorly for MOS, such as the standard-cell, master-slice, or other techniques which depend on automatic placement and routing, have determined the dominance of bipolar technology in mainframes as much for project management as for performance reasons. Meanwhile, MOS technology has exploited the advantages of a cheaper process, simpler circuits, and higher density, to dominate the high-volume, high-complexity applications including particularly storage, microprocessors, and consumer electronics such as watch and calculator chips. Relatively poor performance and the need for handcrafted design appears to be tolerable in this arena.

What happens as the fabrication technology moves to submicron feature sizes is a convergence of the physical and design characteristics of the two technologies. MOS switches attain the excellent physical characteristics of their bipolar counterparts, while bipolar circuits acquire the design problem of sensitivity of circuit operation to parasitic capacitance and layout for which MOS is infamous. There may well be only one submicron technology. The choice of MOS or bipolar switches will have much less effect on design and application characteristics than, for example, whether two or three levels of metal interconnect are used. The analysis above concerning the scaling of MOS circuits indicates that the timing characteristics of the submicron technology will closely resemble those of today's MOS circuits, in that circuit delays are determined mostly by parasitic capacitance -- hence, the length -- of interconnects. The observation of Sutherland and Mead in [3], that it is the wires rather than the switching elements that dominates the cost (area and energy) and performance limitations of microcircuits, is particularly apt for the submicron technology.

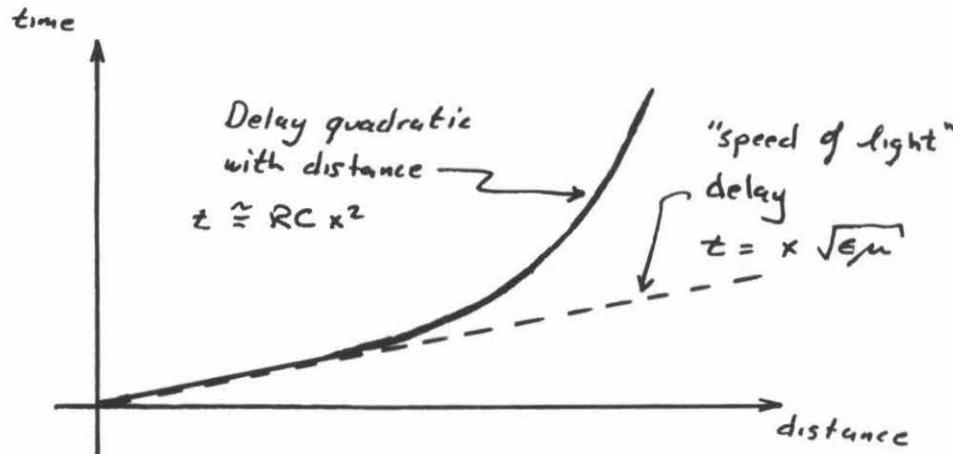
Diffusion Delay in Wires

Wires introduce still another kind of performance limitation as micro-circuit technology is scaled to submicron dimensions. All wires have a non-zero resistivity at ordinary temperatures. This parasitic resistance combined with the parasitic capacitance of the wire limits how well the wire approximates an equipotential. A signal driven onto a wire at one point becomes delayed and its transition time degraded progressively at points more distant from the driven point. The way in which the potential equalizes across the length of a resistive wire is governed by a diffusion equation:

$$RC \frac{dV}{dt} = \frac{d^2V}{dx^2},$$

where V is the potential on the wire as a function of both time (t) and position (x), and R and C are the resistance and capacitance per unit length (the same units in which x is measured). Heat conduction in solids is an example of another diffusion process. The potential equalizes by diffusion along a wire in the same way in which the temperature equalizes across a heat conductor. One can see intuitively that the rate at which a voltage or temperature step propagates decreases with distance from the driven point. The time required for the midpoint of the step to reach a distance x is just RCx^2 , and the step is also "smeared out" in time in proportion to the square of the distance.

As indicated in figure 1, the propagation of signals on wires is limited over short distances by the rate of electromagnetic propagation and over



long distances by diffusion. Please notice that -- to first order, neglecting fringing -- the wire delay is independent of wire width, since the smaller resistivity due to larger cross-section is just balanced by increased capacitance from larger width. The diffusion delay depends on the material resistivity, wire cross-section, and capacitance. What is the magnitude of these delays today, and how do they scale?

Polysilicon cannot be used even in today's MOS technology to distribute

delay-sensitive signals such as clocks more than a mm or so, since it is a material of fairly high resistivity. For distance "x" measured in mm, the parameter RC is for today's typical 5 micron silicon-gate nMOS process about 2 nsec for polysilicon wires (and somewhat variable), about 1 nsec for diffused wires, and only about 0.001 nsec for metal wires. The slowness of polysilicon and diffused wires does not introduce any serious performance limitations at today's feature size, because long runs can be routed on the metal layer(s).

The longest runs on a chip 5 mm on a side need not exceed 10 mm. The delay over this distance would be about 200, 100, and 0.1 nsec respectively for polysilicon, diffused, and metal wires. Light would traverse this same distance in glass similarly to the metal in about 0.1 nsec. Thus the propagation of signals in metal is today not limited by diffusion, but for very long metal wires the resistance is quite close to sufficient to produce critical damping.

How Diffusion Delay in Wires Scales

As was indicated before, the thickness of the various conductive and insulative layers will normally be scaled down together with the plan view dimensions in order to maintain the same relative surface flatness and field geometries. Wire cross-section then scales down quadratically with feature size, and its resistance per unit length scales up quadratically, but this scaling does not significantly change the parasitic capacitance per unit length. If this form of scaling were followed exactly, the factor RC would scale up quadratically. For example, if feature size were scaled down by a factor of 10 to 0.5 micron poly width, the values of RC given above would scale up by a factor of 100. A 10 mm metal wire would exhibit a delay of about 10 nsec. This delay might seem manageable by today's standards, but meanwhile the transit time for such a technology decreases by a factor of 10 from about 0.25 to 0.025 nsec. So, the number of transit times required for the potential to equalize 10 mm across the wire has increased by 1000 times, the third power of the scaling factor, from about 0.4 to 400 transit times. The metal wires are then functionally much like today's diffused or poly wires.

For how long a wire is the diffusion delay equal to one transit time? This way of relating wire delay to switching delay is helpful for seeing at what point one must start paying attention to wire delay. For polysilicon, diffused, and metal wires in today's typical 5 micron process these lengths are approximately 0.3, 0.5, and 17 mm, respectively. For the same geometry scaled down by a factor of 10 to produce an 0.5 micron process, these distances scale down by more than 10, that is, by the 3/2 power of the scale factor, to become about 0.01, 0.02, and 0.5 mm. The potential will equalize across 2 times these distances in 4 transit times, 3 times these distances in 9 transit times, etc.

Design Countermeasures

Since diffusion delay is quadratic with distance, it is possible to reduce the delay on long wires and make the delay linear with distance by interposing repeaters at intervals along a long wire. One can build simple repeater amplifiers whose delay is only 5 to 10 transit times. The optimum

spacing for repeaters is the distance at which the wire delay equals the delay of the repeater. The 10 nsec delay for a 10 mm long metal wire could be reduced to about 3 nsec by interposing repeaters about every 1.5 mm.

Process Countermeasures

There is obviously going to be some pressure on process designers to reduce the value of RC. The resistance per unit length of minimum dimension wires can be achieved only by making the layers thicker or by using material of lower resistivity. However, the approach of increasing the thickness to more than that implied by today's thickness-to-width aspect ratio is quite a bit more difficult from the process standpoint, and eventually becomes self-defeating by increasing the capacitance. Process measures which reduce the parasitic capacitance -- SOS, greater level separation -- benefit both the parasitic delays and the diffusion delay in wires.

Unfortunately, even if such measures could reduce the value of RC by a factor of, say, 4 over simple scaling; the distance over which the potential could equalize in a particular number of transit times increases only by the square root, or 2.

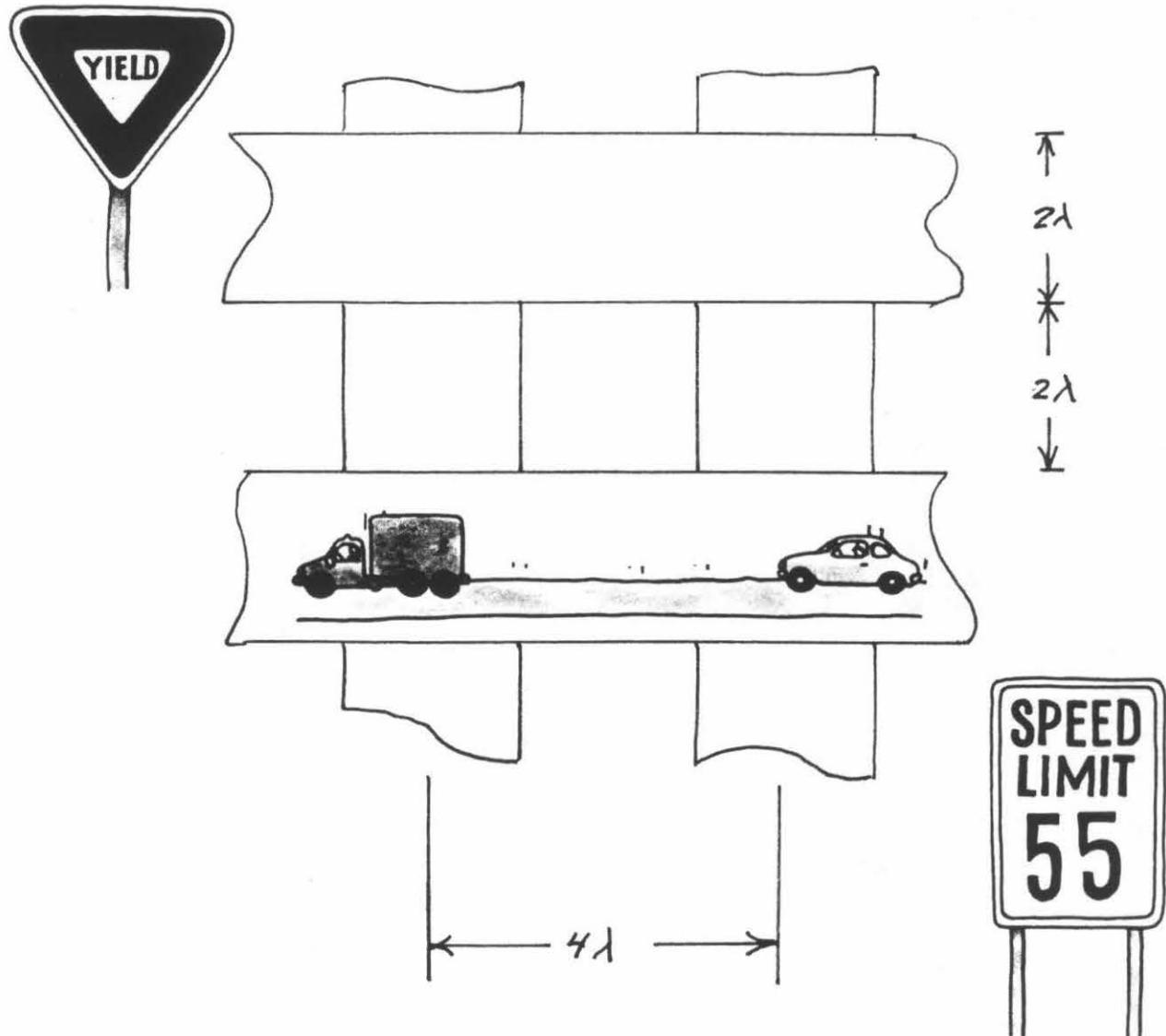
The Trouble with Synchronous Systems

Synchronous systems are most efficient if they operate at clock periods which are relatively small multiples of the transit time. Multiples less than 100 are rarely achieved, but multiples more than about 400 are of questionable efficiency since the combinational parts then operate at an inefficient duty factor, spending less than one percent of their existence switching, and spending the rest holding a static output implied by a static input. When transit time scales down, so should clock periods.

Scaling creates trouble for synchronous systems, not by making it impossible to build them, but by forcing their clock periods to be inefficiently large. This inefficiency comes about in two ways. First, it is desirable for reliability and performance reasons to distribute the clock signal(s) with as little skew as possible. That part of a clock period set aside to allow for skew is not otherwise useful. Clocks distributed on metal show a skew today of less than a transit time, but by the figures above for an 0.5 micron process, it would then be difficult to keep the skew below about 100 transit times. The other difficulty in scaling synchronous systems is the convention that communication between any two points in the system can be achieved within a single clock period, and which results in very large clock periods, particularly if synchrony is to be maintained across chips. Some synchronous designs of the pipeline type avoid this difficulty by limiting communication within a single clock cycle to that between physically local parts.

In Summary, an Analogy

Let me share with you a simple analogy, illustrated in figures 2 and 3, and whose object is to provide another way of looking at the complexity and communication limitations the designer of VLSI systems will encounter. If one takes half of the width of a poly conductor as the length unit λ (the same λ as appears in Mead and Conway [1]), with the scaling of feature size



As λ is scaled down:

Resistance per unit length scales up quadratically

(per λ unit scales up linearly)

Capacitance per unit length constant

(per λ unit scales down linearly)

Time to go x blocks is constant in scaling

Figure 2

represented by changing the value of λ , it turns out that the diffusion delay for a distance measured in λ units is constant. A picture of this world looks not unlike a road network, so we note that the time required to communicate across a certain number of blocks remains constant as λ is scaled.

Figure 3 shows four different stages of the microcircuit technology, with the result of making features smaller and chip areas larger by scaling up the chips to make their blocks similar to city blocks. The complexity of one of today's microprocessors is not unlike a multi-level road network covering the Los Angeles basin at urban density, e.g. not unlike L.A. really is. The next two stages illustrate not only the complexity, but also the communication problem that the time required to cross a chip increases even as the switches become faster. Local autonomy in function and in timing may work on VLSI chips just as it appears necessary in our social and political organization.

References

- [1] Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1979.
- [2] Amr Mohsen, "Devices and Circuits for VLSI", this issue.
- [3] Ivan Sutherland and Carver Mead, "Microelectronics and Computer Science," *Scientific American*, September 1977.

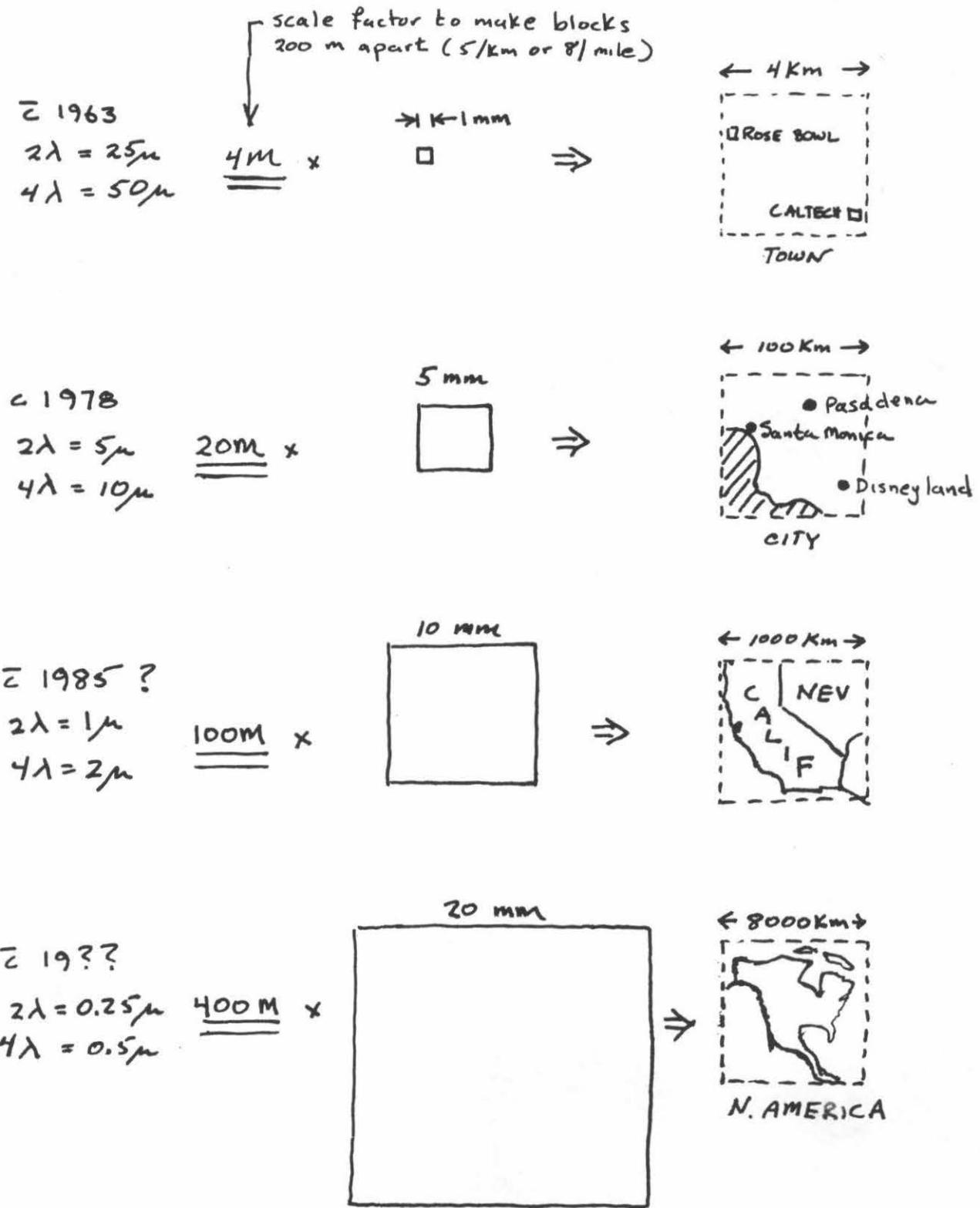


Figure 3

SELF-TIMED LOGIC SESSION

Characterization and Scaling of MOS Flip Flop Performance
in Synchronizer Applications

Thomas J. Chaney and Fred U. Rosenberger
Washington University
St. Louis, Missouri

Abstract:

The measured and calculated values of the Flip Flop parameters needed to specify synchronizer reliability are presented for 3 different depletion-load, silicon gate, NMOS, R-S Flip Flop circuits with gate lengths ranging from $6\mu\text{m}$ to $4.2\mu\text{m}$. Estimates of the probability of synchronizer failure to resolve within allowed or desired times can be determined from these parameters.

This work has been supported by the Division of Research Resources of the National Institutes of Health under Grant RR-00396.

I. Introduction

A fundamental problem exists in communicating between any two concurrently operating digital systems that lack a common time reference. This problem involves the inability to build a completely reliable synchronizer or arbiter that will work in a prescribed amount of time. The problem may be stated as:

Given two independent input events, determine within a bounded time of the arrival of the first one, which of them arrives first. If the difference in arrival time is small, either decision is acceptable but must be made irrevocably within a bounded time interval. By increasing the time allowed, the probability of not being able to determine which arrives first is reduced but this probability cannot be made zero. This lack of decision, or resolution, usually manifests itself by the output of a Flip Flop or output of a Flip Flop containing circuit being either undefined (between a high and low state), oscillating a number of times between the high and low states, or changing states at an arbitrary time after the input events.

It must be noted that the fact that a Flip Flop output is not resolved does not necessarily mean that the system the Flip Flop is embedded in will fail, only that it may fail. Experience has shown that the frequency of system failures may be significantly less than the calculated frequency of Flip Flop not resolved occurrences. Thus the data presented here represents an upper bound on system failures, not a prediction of system failures.

Previously published work has dealt with both the fundamental problem and with circuit characterization, primarily for TTL circuits. (1 through 21) This paper addresses the problem of characterizing the performance of MOS synchronizer Flip Flops in a manner that allows prediction of error probabilities based on Flip Flop parameters and the application conditions. This characterization of the time response of Flip Flops for use as synchronizers or arbiters involves several parameters in addition to the usually specified propagation delay. Section II briefly describes the additional parameters required, Section III gives equations which can be used for calculating these parameters, Section IV gives the results of measurements on several Flip Flops and finally, Section V discusses the implications of scaling on the performance of synchronizers and systems using them.

II. Definition of Synchronizer Flip Flop Parameters

The Flip Flop characterization outlined here is derived from a phenomenological model based directly upon our experimental studies of electric circuits and is taken from the development in Wann, et al. (16). The synchronizer Flip Flop circuit considered in this paper is shown in Figure 1 with both R and S inputs initially high. We wish to estimate the probability that the synchronizer will fail to achieve a logically defined and stable output by a time t' after the time the earlier of the two inputs have switched low which we shall assume occurs at time $t=0$. Let us further assume that the high to low transition at the Reset input takes place at a time t_d relative to the Set input transition and that t_d has a uniform probability density $p(t)$ over an interval $t_a \leq t_d \leq t_m$, that is

$$p(t_d) = \begin{cases} 0 & t_d < t_a \\ \frac{1}{t_m - t_a} = \frac{1}{\delta} & t_a \leq t_d \leq t_m \\ 0 & t_m < t_d \end{cases} \quad (1)$$

where t_m is sufficiently positive that the Flip Flop always Resets, and t_a is sufficiently negative that the Flip Flop always Sets, within the normally specified propagation delay time following the time the earlier of the two inputs has switched low.

For t_d uniformly distributed over this interval, we can define $F(t')$ as the probability that the Flip Flop output has not achieved a logically defined and stable value at the time t' for a single occurrence of the Set and Reset inputs going low. Experiments have shown that for sufficiently large values of t' ($t' > h$) this probability can be approximated by

$$F(t') = \frac{T_o}{\delta} \exp \frac{-t'}{\tau_r} \quad t' > h \quad (2)$$

where $\delta = t_m - t_a$

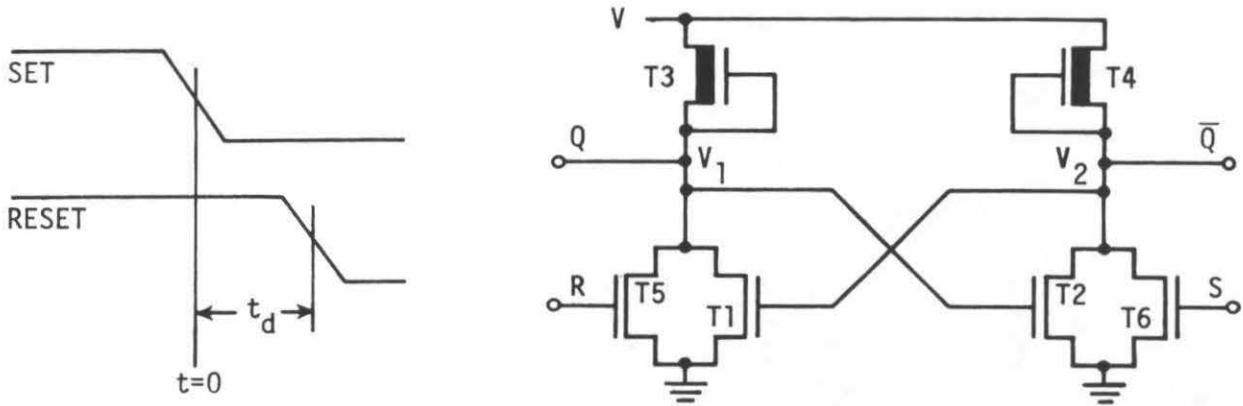
and the parameters τ_r and T_o depend upon the specific circuit. Multiplying both sides of Equation (2) by δ and taking the natural logarithm of both sides yields

$$\ln[\delta F(t')] = -\frac{1}{\tau_r} (t') + \ln [T_o] \quad t' > h \quad (3)$$

Hence, a semilogarithmic plot of $\delta F(t')$ versus t' , as shown in Figure 2, is a straight line of slope $(-1/\tau_r)$. By setting t' equal to zero in Equation

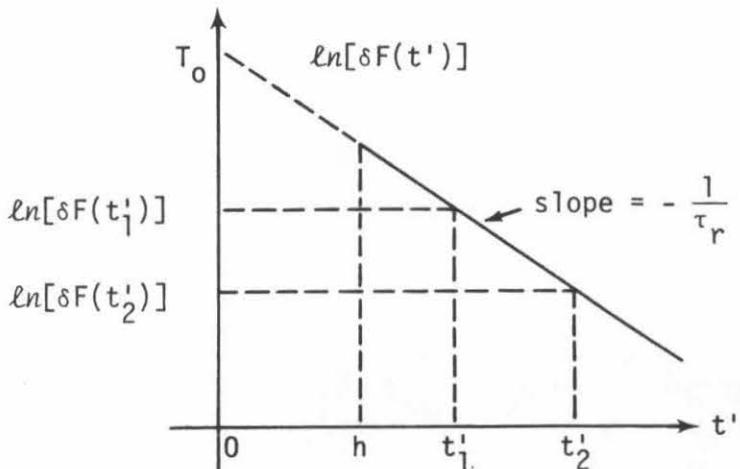
(3) it is clear that T_o is equal to $\delta F_e(0)$ where $F_e(t')$ is the linear extrapolation of $F(t')$ for $t' \leq h$.

The synchronizer Flip Flop thus can be characterized by three parameters, τ_r , T_o , and h . Clocked Flip Flop types such as J-K or D Flip Flops can be characterized in a similar manner where t' is the time after the clocking event. τ_r and T_o are calculated and the results compared with measurements for NMOS Flip Flops in Sections III and IV.



NMOS R-S FLIP-FLOP

FIGURE 1



GRAPHICAL REPRESENTATION OF PROBABILITY THAT SYNCHRONIZER HAS NOT ACHIEVED A LOGICALLY DEFINED AND STABLE STATUS

FIGURE 2

III. Calculation of values for T_o and τ_r

In this section equations are presented for the values T_o and τ_r as a function of the circuit parameters. The goal is to produce relatively simple equations in terms of the MOSFET threshold voltages and dimensions. More complete and precise equations can obviously be developed but simulation would probably be a more reasonable approach if greater accuracy is desired. It should be noted, however, that the simulation of Flip Flops in the metastable region with general purpose simulators should be approached with caution since such a simulation may be more a test of the numerical analysis techniques used in the simulator than of the circuit being simulated. The circuit used for the analysis is shown in Figure 1. The analysis is divided into two parts, the initialization time, t_i , starting when the Set and Reset inputs go low and lasting until the Flip Flop outputs reach the metastable state and the resolution time, τ_r , starting from this point and lasting until the outputs diverge.

We are interested in the case where the Set and Reset signals go low nearly simultaneously so that a short time after they both go low the two outputs are approximately equal and at the same voltage as an inverter with its input connected to its output. We will call this voltage V_{INV} . If the circuit were balanced with perfect symmetry, the two output voltages exactly equal and the circuit noise-free, the Flip Flop would remain in this balanced condition indefinitely. Any initial imbalance will be amplified by the two inverters forming the Flip Flop and will eventually cause the outputs to reach the normal high and low voltages. The time required to reach the normal output levels is dependent on the initial imbalance in the Flip Flop outputs and on the gain and frequency characteristics of the inverters forming the Flip Flop. We will first determine the resolving time, τ_r , the time to reach defined output levels based on an initial small difference in output voltages, and then determine the difference in output voltages at the beginning of the resolving time period as a function of the relative time that the Set and Reset inputs go low. Obviously t_i plus τ_r equal the t' defined in Section II. Figure 3 shows a typical set of waveforms and the definition of the initialization time and resolving time for a Flip Flop.

For the resolving time calculation we will use a simple linear model since the major part of the operation occurs with V_1 and V_2 confined to a narrow voltage range close to V_{INV} . Toward the end of the resolving time, as V_1 and V_2 approach the normal high and low levels, this linear approximation becomes less and less valid but the error introduced by using the linear approximation is small.

In addition to errors introduced by the linear model at the end of the resolving time, the definition of when the outputs are resolved affects the calculations. As will be seen, the definition of the voltage at which the output is resolved and the use of linear approximations for the circuits will affect the calculated values for T_o but not the value of τ_r .

The pulldown transistors are obviously operating in the saturation region when their inputs and outputs are equal to V_{INV} since they have a positive threshold voltage, V_T , but the pullup transistor may be operating in either the saturation or resistive region. We will assume here that the pullup threshold voltage is sufficiently negative that it is in the resistive operating region when its drain voltage is at V_{INV} . Somewhat simpler equations are obtained if the pullup is assumed to be operating in the saturation region instead.

Figure 4 shows an equivalent circuit for the Flip Flop in the linear operating region when both the Set and Reset inputs are low, the pulldown transistor is saturated, and the pullup transistor is in the resistive region. For this case:

$$I_{T1} = \frac{\mu C_G}{2 \cdot L^2} (V_2 - V_{TPD})^2 \quad (4)$$

$$I_{T2} = \frac{\mu C_G}{2 \cdot L^2} (V_1 - V_{TPD})^2 \quad (5)$$

C_{TOT} = Total capacitance of node 1 including the gate capacitance of T2

$$R_1 = \left. \frac{dV_{DS}}{dI_{DS}} \right|_{V_{DS}=V_{DD}-V_{INV}} = \frac{L^2 \cdot k}{C_G \cdot \mu} \cdot \frac{1}{(V_{INV} - V_{DD} - V_{TPU})} \quad (6)$$

$$V_{INV} = \frac{k \cdot V_{TPD} + V_{DD} + V_{TPU}}{1+k} + \sqrt{\frac{V_{TPU}^2 + 2k \cdot V_{TPU} V_{TPD} - k \cdot V_{TPD}^2 + 2k \cdot V_{DD} (V_{TPD} - V_{TPU}) - k \cdot V_{DD}^2}{1+k}} \quad (7)$$

where: μ = Mobility of electrons

C_G = Gate capacitance of pulldown transistor

L = Gate length of pulldown transistor

$k = \frac{L_{PU}}{W_{PU}} \cdot \frac{W_{PD}}{L_{PD}}$ where L and W are the length and width of the PU (pullup) and PD (pulldown) transistors

V_{TPD} = threshold voltage of pulldown transistor

V_{TPU} = threshold voltage of pullup transistor

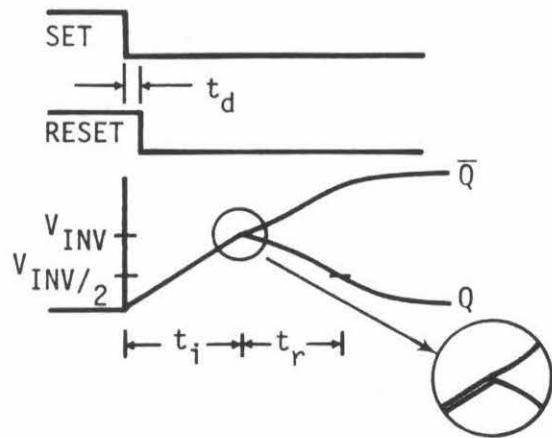
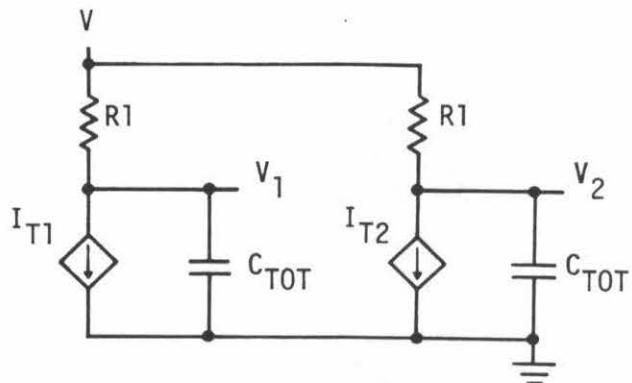
TYPICAL WAVEFORMS FOR t_d SMALL AND POSITIVE

FIGURE 3



EQUIVALENT CIRCUIT FOR FLIP FLOP IN THE METASTABLE REGION

FIGURE 4

If the circuit has voltages $V_1 = V_{INV} - \Delta V$ and $V_2 = V_{INV} + \Delta V$ at time t_i , then the voltage V_1 as a function of time is given by

$$V_1 = V_{INV} - \Delta V \cdot \exp[p(A-1)(t-t_i)] \quad t \geq t_i \quad (8)$$

where $p = \frac{C_G \cdot \mu(V_{INV} - V_{DD} - V_{TPU})}{C_{TOT} \cdot L^2 \cdot k}$ = the bandwidth of one inverter stage of the Flip Flop

$$A = k \cdot \frac{V_{INV} - V_{TPD}}{V_{INV} - V_{DD} - V_{TPU}} = \text{magnitude of the low frequency gain of one inverter stage}$$

thus the τ_r defined in Section II is given by

$$\boxed{\tau_r = \frac{1}{p(A-1)} = \frac{C_{TOT} L^2}{C_G \mu} \cdot \frac{k}{(V_{INV} - V_{TPD}) - (V_{INV} - V_{DD} - V_{TPU})}} \quad (9)$$

$$\text{and } V_1 = V_{INV} - \Delta V \cdot \exp \frac{t-t_i}{\tau_r} \quad (10)$$

$$V_2 = V_{INV} + \Delta V \cdot \exp \frac{t-t_i}{\tau_r} \quad (11)$$

If the magnitude of the gain in the metastable region is large, τ_r is approximately equal to the inverse of the gain-bandwidth product. If the

magnitude of the gain is close to 1, τ_r becomes very large, but logic inverters with a gain of one are not very desirable for other reasons.

Next we wish to find the initial offset voltage ΔV as a function of the time difference between the Set and Reset signals, t_d . We will assume that both V_1 and V_2 are initially at 0 volts and start to increase linearly when the Set and Reset inputs, respectively, go low, and continue going positive until the average voltage of V_1 and V_2 is equal to V_{INV} . The difference between V_1 and V_2 at this time will be equal to $2 \cdot \Delta V$. Obviously in the real circuit the voltages V_1 and V_2 do not increase linearly once they are greater than V_{TPD} or when the pullup transistor is no longer saturated. Also the values of the two outputs, V_1 and V_2 , start to diverge before their average reaches V_{INV} , but for ease of analysis we will consider the two regions of operation to be distinct, one in which the initial difference between the two outputs is established, and second the resolving time during which this small initial difference is amplified until the outputs reach the levels we establish as resolved. If we assume that the pullup transistors remain saturated, the pullup current is

$$I_{PU} = \frac{C_{GPU} \cdot \mu}{2 \cdot L_{PU}^2} (V_{TPU})^2 = \frac{C_G \cdot \mu}{2 \cdot L^2 \cdot k} (V_{TPU})^2 \quad (12)$$

$$\text{let } w = \frac{dV_1}{dt} = \frac{dV_2}{dt} = \frac{I_{PU}}{C_{TOT}} = \frac{C_G \cdot \mu}{C_{TOT} \cdot 2 \cdot k \cdot L^2} (V_{TPU})^2 \quad (13)$$

$$\text{then } 2 \cdot \Delta V = t_d \cdot w$$

$$\text{or } \Delta V = \frac{t_d \cdot w}{2}$$

$$\text{and } t_i = \frac{|t_d|}{2} + V_{INV} \cdot \frac{1}{w} \quad (14)$$

If we define the Flip Flop to be resolved when one of the outputs reaches $V_{INV}/2$ or halfway between 0 volts and V_{INV} we find the probability that the Flip Flop output is not resolved at t' is:

$$F(t') = \Pr[\text{Flip Flop output not resolved at } t']$$

$$F(t') = \begin{cases} \Pr[1/2 V_{INV} < V_1(t')] & t_d > 0 \\ \Pr[1/2 V_{INV} < V_2(t')] & t_d < 0 \end{cases} \quad (15)$$

Substituting equations (10) and (11) for V_1 and V_2 :

$$F(t') = \begin{cases} \Pr[1/2 V_{INV} < V_{INV} - \frac{t_d + w}{2} \exp \frac{t' - t_i}{\tau_r}] & t_d > 0 \\ \Pr[1/2 V_{INV} < V_{INV} + \frac{t_d + w}{2} \exp \frac{t' - t_i}{\tau_r}] & t_d < 0 \end{cases} \quad (16)$$

$$F(t') = \begin{cases} \Pr[t_d < \frac{V_{INV}}{w} \exp \frac{|t_d|}{2\tau_r} \cdot \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})] & t_d < 0 \\ \Pr[t_d > \frac{V_{INV}}{w} \exp \frac{|t_d|}{2\tau_r} \cdot \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})] & t_d > 0 \end{cases} \quad (17)$$

$$F(t') = \Pr[|t_d| < \frac{V_{INV}}{w} \exp \frac{|t_d|}{2\tau_r} \cdot \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})] \quad (18)$$

Equation (18) is valid only for $t' > h$. For h large enough $t_d \ll \tau_r$ so that

$$\exp \frac{|t_d|}{2\tau_r} \approx 1$$

$$F(t') = \Pr[|t_d| < \frac{V_{INV}}{w} \cdot \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})] \quad (19)$$

If t_d is uniformly distributed over δ , δ includes the range of t_d , and t' is large enough that

$$\delta > \frac{V_{INV}}{w} \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r}) \quad (20)$$

then from (19); $F(t')$ is the portion of the interval, δ , for which $|t_d|$ is less than $\frac{V_{INV}}{w} \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})$, or is twice $\frac{V_{INV}}{w} \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r})$ divided by δ .

Thus $F(t') = \frac{2V_{INV}}{\delta \cdot w} \exp (\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r}) \quad (21)$

or

$$\delta F(t') = \frac{2V_{INV}}{w} \exp\left(\frac{V_{INV}}{w \cdot \tau_r} - \frac{t'}{\tau_r}\right) \quad (22)$$

and

$$T_o = \frac{2 \cdot V_{INV}}{w} \exp\left(\frac{V_{INV}}{w \cdot \tau_r}\right). \quad (23)$$

IV. Measurement of Synchronizer Flip Flop Parameters

A. Test method

The method used to determine experimentally a plot of the type shown in Figure 2 involves observing $\delta F(t')$ for at least two distinct times t'_1, t'_2, \dots, t'_n that are greater than h . The requirement that the input event time t_d is uniformly distributed over the interval $[t_a, t_m]$ is achieved experimentally by obtaining both input signals from the same timing source, and slowly adjusting the delay of one input signal so that in the course of an experiment a large set of values of t_d , uniformly distributed over the interval $[t_a, t_m]$, are generated.

A simplified drawing of the test setup used to measure $\delta F(t'_n)$ is shown in Figure 5. Note that the sampling clock signal is derived only from the Set input to the F-F under test, not the "earlier to the two inputs" defined in Part II. The change in differential delay, t_d , between the two inputs required to change the output, Q, from always remaining low to always switching high is less than 0.1 nsec for the circuits tested. The values of D4, D5, D6 are approximately 5nsec. Thus the error introduced in t'_n by measuring from the Set input transition is very small. The "unstable state detector" shown in Figure 5 provides an output during the period the F-F under test is unresolved. This circuit is implemented with comparator circuits that measure when the Flip Flop outputs are not high or low. (7, 11, 21) The change in differential delay, t_d , is implemented with a 50 ohm adjustable coaxial airline (G.R. type 874-LAL) that is extended and contracted at a constant rate by a leadscrew and motor with a change of t_d of about 10^{-12} seconds per second.

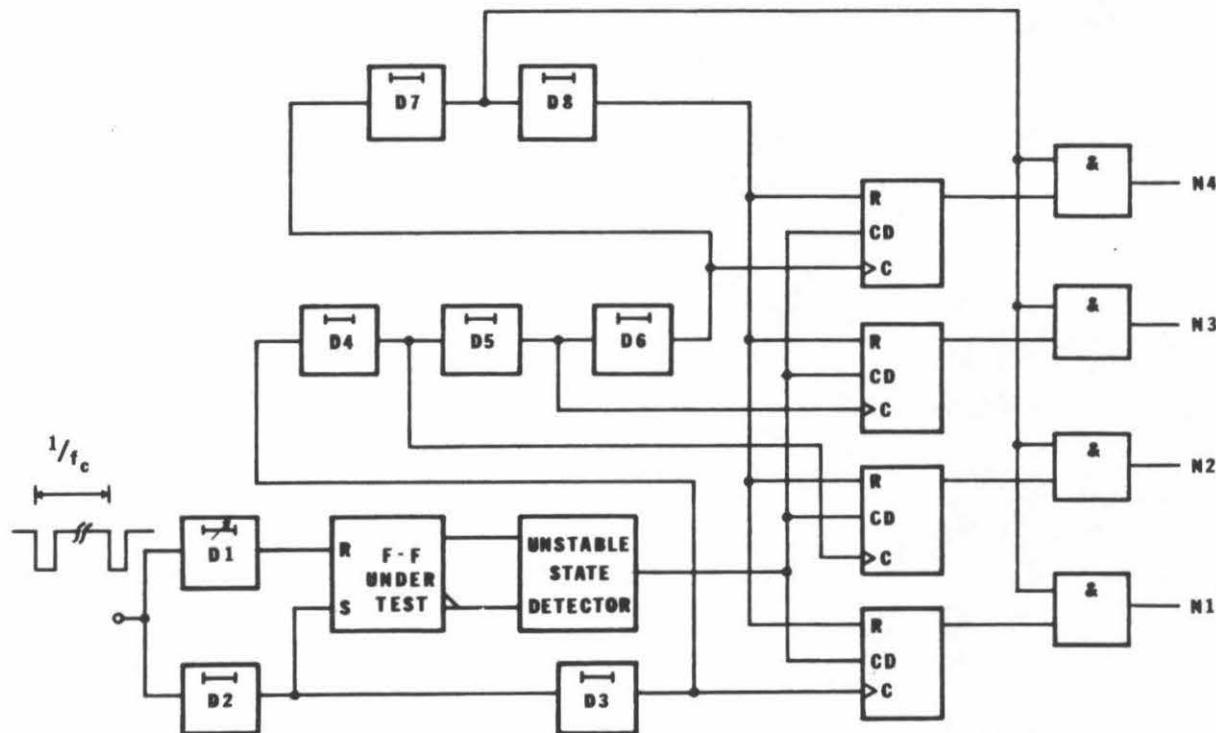
The equation required to calculate values of $\delta F(t'_n)$ for each t'_n from the values collected using the test setup of Figure 5 is based on $F(t'_n) = N_n/N_o$. N_o is the total number of uniformly distributed events ($t_a \leq t_d \leq t_m$) and N_n is the number of times the Flip Flop under test is still unresolved at t'_n . Let f_c be the frequency of the timing source and t_x be the time an experiment runs. Then $N_o = f_c \cdot t_x$ and

$$\delta F(t'_n) = \frac{N_n \cdot \delta}{f_c \cdot t_x}.$$

δ is the total change in differential delay during the experimental run and can be expressed as $\delta = A_{\text{coax}} \cdot t_x$ where A_{coax} is the change in the amount of delay of the adjustable airline per second of experimental run. So:

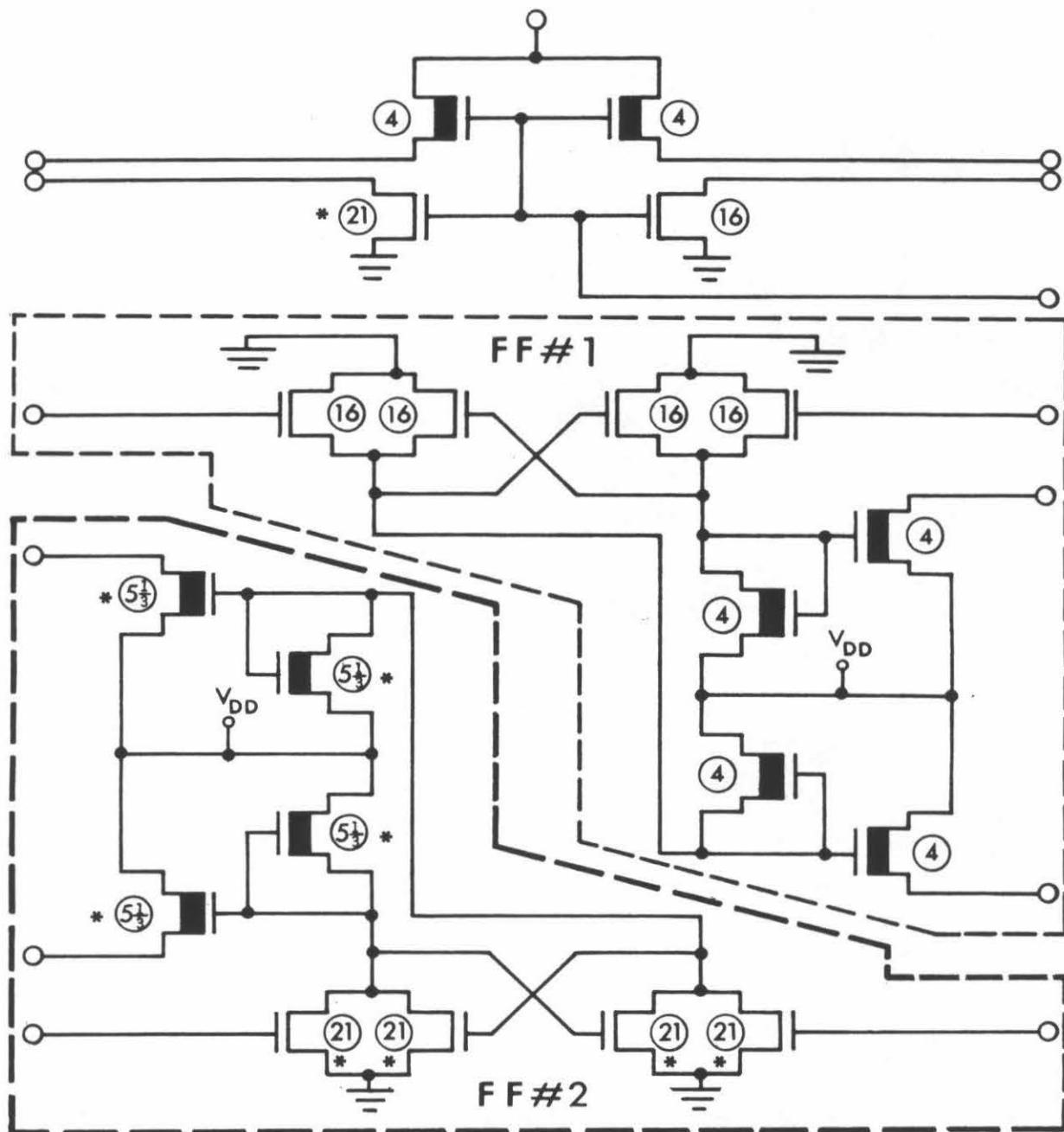
$$\delta F(t'_n) = \frac{A_{\text{coax}}}{f_c} N_n.$$

Thus before an experiment, the event input frequency, the value of A_{coax} , and the values of t'_1 , t'_2 , t'_3 , and t'_4 (indicated in Figure 2) are measured and recorded. Then after several test runs, half with the adjustable delay line extending and half with the delay line contracting, the average values of N_1 through N_4 are used to calculate the values of $\delta F(t'_n)$. The value of h is then determined based on the linearity of the resulting plot or set at smallest value of t'_1 , used during the tests, whichever is larger.



SIMPLIFIED PROBABILITY UNRESOLVED TEST CIRCUIT

FIGURE 5



NOTES: (NO.) IS W/L

* TRANSISTORS HAVE $L = 4 \frac{1}{6} \mu\text{m}$
OTHER TRANSISTORS HAVE $L = 5 \mu\text{m}$

FLIP FLOP CIRCUIT TESTED
FIGURE 6

B. Circuits tested

Two or three circuits from each of 3 different layouts have been tested. All 3 layouts are depletion-load, silicon gate, NMOS, R-S type Flip Flop circuits. Two of the Flip Flop circuits (three of each circuit were tested) are shown in Figure 6. FF #1 and FF #2 are identical, including layout pattern, except for the transistor gate lengths which are 5 μ m and 4 1/6 μ m (mask dimension). The width to length ratio of the Flip Flop transistors is large to minimize the effect of stray capacitance. Four test transistors were included to allow measuring transistor parameters. Note that the outputs of each Flip Flop are buffered with source-follower transistors. During the tests, the source-follower transistor outputs were biased with a 300 ohm load resistor to ground. The source-follower signals, which were approximately 0.4V in amplitude and delayed approximately 5 ns, allowed monitoring the Flip Flop output waveforms with minimum loading.

The third Flip Flop (two circuits were tested) is the arbitration Flip Flop in an arbiter circuit designed by Dr. Ivan Sutherland and was fabricated with 6 μ m gate lengths (mask dimensions). (21) Table 1 gives a comparison of the basic circuit parameters for the 3 circuits tested. The tolerances noted are an indication of the consistency of the measurements and the fitting of the transconductance curves to the simplified equations used to calculate some of the table values.

The measured and calculated values for T_o and τ_r using the nominal values given in Table 1 and equations (9) and (23) are reasonably close as Table 2 shows, especially considering the model used and inability to measure some of the circuit parameters of the Flip Flops tested. It should be noted that the experimental results were not all measured on circuits from the same wafer. Figure 7 shows a graph of mean time between unresolved events for one of the Flip Flops (FF #1) as a function of allowed settling time, event rate, and logic delay. The curve labeled D=1, E=0 depicts the synchronizer Flip Flop performance if the input interrupt event rate is equal to the clock rate and the synchronizer Flip Flop output is tested at the next clock time with no logic gates or delay in series with the Flip Flop output. The curve labeled D=10⁻³, E=0.5 depicts the synchronizer Flip Flop performance if the event rate is one thousandth the clock rate and the synchronizer Flip Flop output, after passing through logic gates or delay equal to one half the clock period, is tested at the next clock time.

CIRCUIT	L μM (2)	k (2)	$V_{DD} = 5.0\text{V}$			C_G (pf _d) (4)	C_{TOT} (pf _d) (4)	μ (5, 6)
			V_{TPD} (6)	V_{TPU} (6)	V_{INV}			
ARBITER -1	6	5	-	-	-	0.087	0.34	-
ARBITER -2	"	"	-	-	-	"	"	-
FF #1-1	5	4	.5V (3)	-3V (3)	2V (3)	0.16	0.40	550 (3)
FF #1-3, 4 (1)	"	"	.5V±.2V	-3V±.6V	2V±.1V	"	"	550±150
FF #2-2	4 1/6	4	.5V (3)	-3V (3)	2V (3)	0.13	0.36	550 (3)
FF #2-3, 4 (1)	"	"	.5V±.2V	-3V±.6V	2V±.1V	"	"	550±150

- (1) Chips 3 and 4 were near neighbors. The test results were all essentially identical.
 (2) Mask dimensions.
 (3) Estimated as average of chip 3 and 4.
 (4) Estimated using layout masks and: [from Ref. (20)]
 Gate - Channel cap. = 4×10^{-4} pf_d/ μm^2
 Diffusion cap. = 0.8×10^{-4} pf_d/ μm^2
 Polysilicon cap. = 0.4×10^{-4} pf_d/ μm^2
 Metal cap. = 0.3×10^{-4} pf_d/ μm^2
 (5) Cm²/Volt Sec.
 (6) Calculated from test transistor transconductance curves.

BASIC CIRCUIT PARAMETERS

TABLE 1

CIRCUIT	MEASURED (NSEC.)				CALCULATED (NSEC.)		$\frac{\tau_r \text{ calc.}}{\tau_r \text{ meas.}}$
	τ_r	T_o	t_{pd} L-H	$\leq h$	τ_r	T_o	
ARBITER -1	2.4	-	-	-	-	-	-
ARBITER -2	2.1	-	-	-	-	-	-
FF #1-1	1.67	20	4	11	.76	58	46%
FF #1-3, 4	1.45	13	4	9	"	"	52%
FF #2-2	1.44	8	-	11	.58	46	40%
FF #2-3, 4	1.20	15	5	9	"	"	48%

EXPERIMENTAL AND CALCULATED VALUES OF SYNCHRONIZER PARAMETERS

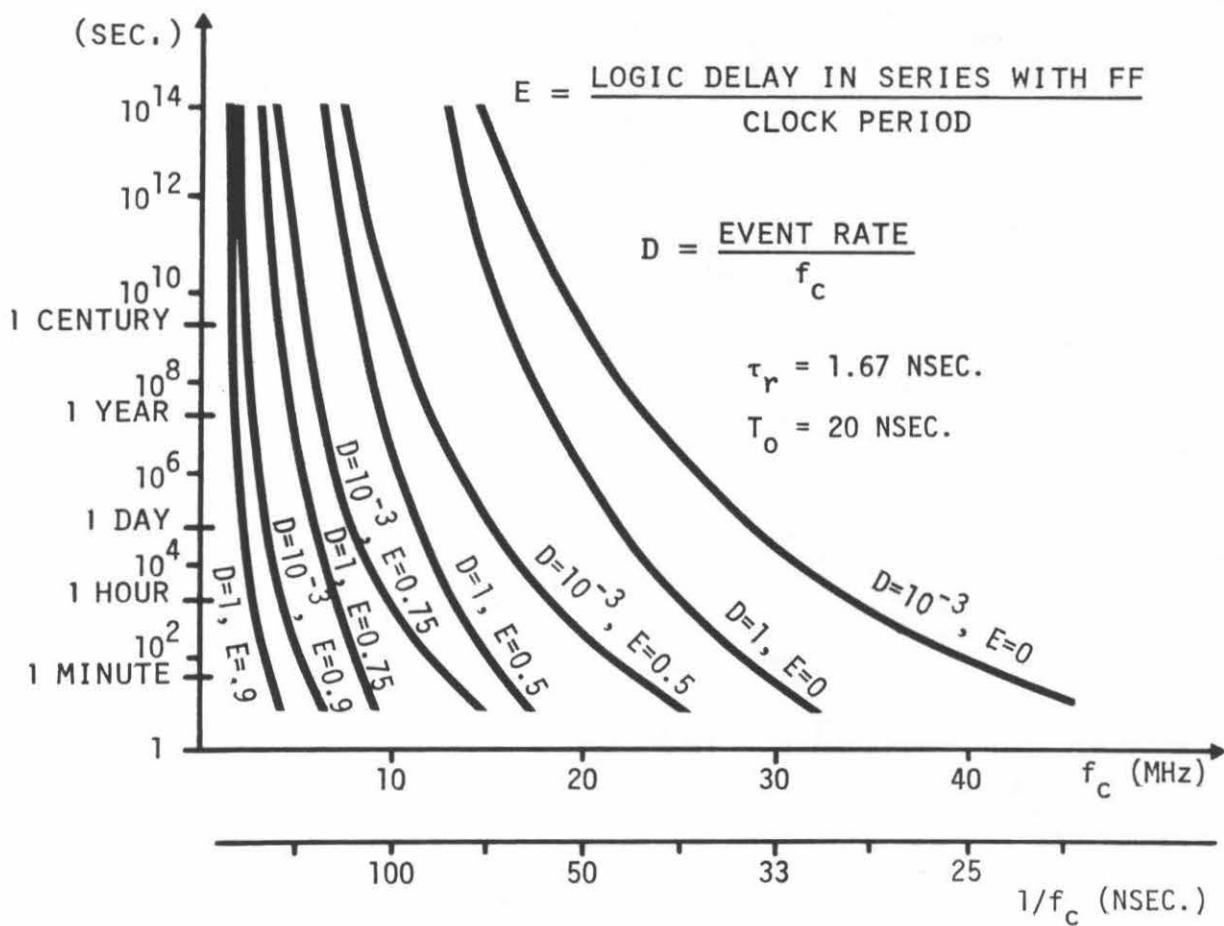
TABLE 2

SELF-TIMED LOGIC SESSION

$$MTBSU = \frac{\exp\left(\frac{1-E}{f_c \cdot \tau_r}\right)}{(T_0)(f_c)^2 D}$$

MEAN TIME BETWEEN
SYNCHRONIZER UNRESOLVED
AT CLOCK TIME (MTBSU)

FOR EVENT OCCURRENCES
UNIFORMLY DISTRIBUTED
WITH RESPECT TO CLOCK



PROBABILITY OUTPUT OF SYNCHRONIZER FLIP FLOP
AND LOGIC IS STILL UNRESOLVED AT NEXT CLOCK TIME

FIGURE 7

V. Conclusions

As feature sizes and voltages are scaled down by α it can be shown using equations (9) and (23) that both T_0 and τ_r are reduced proportionally to the scaling. Since normal propagation delays also scale proportionally to dimensions and voltages (20, 22), the ratio of T_0 and τ_r to propagation delay remains constant. Thus, if system dimensions and operation times are scaled, the probability of a synchronizing Flip Flop failing to resolve within the allowed time remains the same for each occurrence. However, since the operation times are reduced by scaling, a higher rate of unresolved outputs is produced if we take advantage of the ability to perform more operations per second. Also, if we take advantage of the scaling to build systems with α^2 times as many elements and build them with the same proportion of synchronizers, then scaled systems will have α^3 as many synchronizing events per second, all with the same probability of failure as the unscaled system. Thus, if the unscaled system had a mean time between synchronizer unresolved (MTBSU) of one per year, a system scaled by 10 would have a MTBSU of about 9 hours, a significant reduction. Obviously the scaling of MTBSU is not very desirable. The moral here is that the same design techniques for interconnecting subsystems that are used with today's designs and feature sizes may not be directly applicable with scaled circuits.

Considerable refinement in measurements, techniques and sample size, and calculations is possible and some additional effort seems worthwhile. In particular measurements on circuits whose fabrication parameters are better characterized would be worthwhile. In addition, better characterizations of Flip Flop performance with input switching and conditions that cause resolving times between t_{pd} and h is warranted, both to provide better understanding of the circuit operation and for use in asynchronous arbiter circuits where operations are delayed only until the Flip Flop resolves instead of providing a long fixed waiting time. In these arbiter circuits the parameter of interest is expected settling time, not the time required to reduce $F(t')$ to less than some particular value.

VI. References

1. Krevden, N.L., "The Dynamics of the Toggle Action," Proceedings of the Western Joint Computer Conference, Los Angles, CA.: 46-50, May 6-8, 1958.
2. Gray, H., Digital Computer Engineering, Prentice-Hall, Inc., Englewood Cliffs, NJ: 198-201, 1963.
3. Catt, I., "Time Loss Through Gating of Asynchronous Logic Signal Pulses," IEEE Trans. on Electronic Computers, EC-15:108-111, Feb. 1966.
4. Littlefield, W.M., and Chaney, T.J., "The Glitch Phenomenon," Computer Systems Laboratory, Washington University, St. Louis, MO, Technical Memorandum #10, Dec. 1966.
5. Mars, P., "Study of the Probabilistic Behavior of Regenerative Switching Circuits," Proc. IEE, (England), 115:642-668, May 1968.
6. Couranz, G.R., An Analysis of Binary Circuits Under Marginal Triggering Conditions, Computer Systems Laboratory, Washington University, St. Louis, MO, Technical Report #15, Nov. 1969.
7. Adams, R.L., Castaldo, D.R., and Kurz, G.W., "Real-time Detection of Latch Resolution Using Threshold Means," U.S. Patent 3,515,998, June 1970.
8. Wheeler, D.J., "System Design for Non-Engineering Students and the Synchronization Problem," Proc. of the Conf. on Teaching of Computer Design, Univ. of Newcastle-upon-Tyne, England, 88-94, Sept. 1971.
9. Plummer, W.W., "Asynchronous Arbiters," IEEE Trans. on Computers, C21:37-42, Jan. 1972.
10. Workshop on Synchronizer Failures, Washington University, St. Louis, MO, April 27-28, 1972.
11. Chaney, T.J., Ornstein, S.M., and Littlefield, W.M., "Beware the Synchronizer," in Digest of Papers - COMPON '72, IEEE Computer Society Conference: 317-319, Sept. 1972.
12. Chaney, T.J., and Molnar, C.E., "Anomalous Behavior of Synchronizer and Arbiter Circuits," IEEE Trans. on Computers, C-22:421-422, April 1973.
13. Kinnement, D.J., and Edwards, D.B.G., "Circuit Technology in a Large Computer System," The Radio and Electronic Engineer, 43:435-441, July 1973.

14. Chaney, T.J., The Synchronizer 'Glitch' Problem, Computer Systems Laboratory, Washington University, St. Louis, MO, Technical Report #47, Feb. 1974.
15. Hurtado, M., Dynamic Structure and Performance of Asymptotically Bistable Systems, D.Sc. Dissertation, Dept. of Electrical Engineering, Washington University, St. Louis, MO, 1975.
16. Wann, D.F., Molnar, C.E., Chaney, T.J., and Hurtado, M., "A Fundamental Problem Associated with the Physical Realization of Certain Classes of Petri-Nets," Conf. on Petri-Nets and Related Methods, MIT, Cambridge, MA, S.S. Patil-Editor, July 1975, to be published. Copies available as Technical Memorandum #215, Computer Systems Laboratory, Washington University, 724 S. Euclid Ave., St. Louis, MO 63110.
17. Pechoucek, M., "Anomalous Response Times of Input Synchronizers," IEEE Trans. on Computers, C-25, No. 2:133-139, Feb. 1976.
18. Kinniment, D.J., Woods, J.V., "Synchronization and Arbitration Circuits in Digital Systems," Proc. of the IEE, (England), Vol. 123, No. 10:961-66, Oct. 1976.
19. Marino, L.R., "The Effect of Asynchronous Inputs on Sequential Network Reliability," IEEE Trans. on Computers, C-26, No. 11:1082-90, Nov. 1977.
20. Mead, C.A., and Conway, L.A., Introduction to VLSI Circuits, to be published.
21. Sutherland, I.E., "A Specific Arbiter Design, ARBIT2," Computer Science Department, California Institute of Technology, Pasadena, CA, Display File No. 226, Sept. 22, 1976.
22. Dennard, R.H., Gaensslen, F.H., Yu, H.N., Rideout, V.L., Bassous, E., and LeBlanc, A.R., "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," IEEE Journal of Solid-State Circuits, SC-9, No. 5, Oct. 1974.

Synchronization Strategies

M. J. Stucki and J. R. Cox, Jr.
Department of Computer Science
Washington University
St. Louis, Missouri

This work has been supported in part by the Division of Research Resources
of the National Institutes of Health under Grant RR 00396

INTRODUCTION

Computing systems are now frequently composed of independently clocked subsystems that cooperate to perform the function desired for the whole. This type of architecture has many advantages and promises to be the standard for the foreseeable future. With the trend towards more and more gates per chip, the number of chips per subsystem gets smaller and smaller, and we can expect to soon see one or more subsystems per chip. This transition will require contributions from disciplines previously outside the field of chip design, and every issue will have to be carefully worked out beforehand because debugging chips of this complexity is a difficult and costly task.

This paper addresses one of those issues - the design of reliable synchronization logic for interfacing independently clocked subsystems. The design of this logic is not a normal exercise in clocked logic design because the operating environment is such that the response times of some flipflops will be unbounded, and an improper appreciation of this phenomenon can result in designs plagued by intermittent synchronization failures. The absence of a bound has been documented in the literature, but only from an experimental and analytic standpoint, and no generally applicable methodology for dealing with it has been suggested. As a result, it is not common knowledge among logic designers, and future systems are liable to suffer from it. The objective of this paper is to assist the logic designer by reviewing the basic phenomenon, characterizing it quantitatively, and presenting techniques for coping with it.

THE SYNCHRONIZATION PROBLEM

The specification sheet for a sequential device gives operating constraints such as setup times, hold times and maximum clock rates. These are constraints that must be met in order to assure a consistent interpretation of the input signals. Consider, for example, the common type D flipflop. If its data input value is logically defined during the specified setup and hold periods, the value seen by the flipflop when clocked will be unambiguous, and two or more flipflops presented with the same conditions will see the same value. However, if the constraints are not met, then interpretations may differ. For example, one flipflop could interpret a logically undefined input value as a 1 while another flipflop could interpret the same signal as a 0. Similarly, if the input signal is changing value, one flipflop could capture the before value while another flipflop could capture the after value. Consistent interpretations are guaranteed only for cases that meet the specified input constraints.

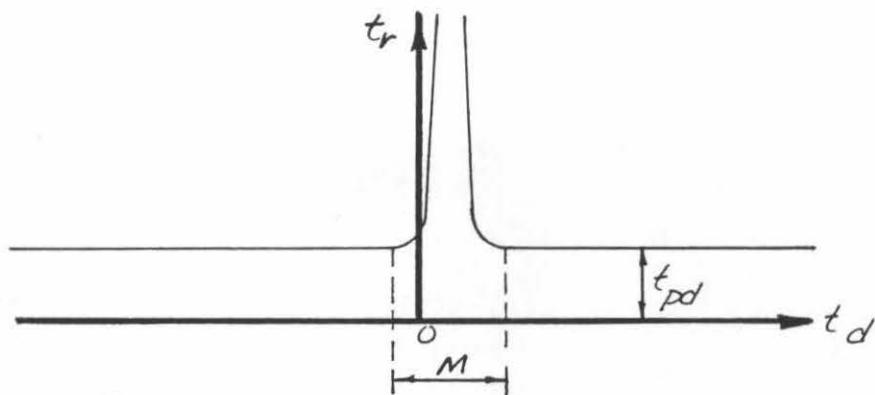
Preserving consistency is a primary concern in the design of systems, and the proper functioning of a system requires that all devices that depend on the value of a given signal at a given time see the same value, i.e., they all see a 0 or they all see a 1. The logic designer satisfies this requirement by trying to assure that the input constraints are met for each device. In

cases where he cannot adequately control an input signal, for example a signal originating external to the system, he solves the consistency problem by allowing only one observer and therefore only one interpretation. The traditional method is to gate the signal into a flipflop and then use the flipflop's output value as the signal to be processed by the system. The flipflop is said to synchronize the input signal because it gives the designer a copy which he can control.

In point of fact, the copy is not totally under the designer's control. Although it is common knowledge that violating the input constraints of a flipflop can cause unusual behavior and although it is common practice to allow for a longer than normal propagation delay in these applications, experimental studies [1, 2, 3, 4, 5, 6] and theoretical studies [7, 1, 8] indicate that the response times are actually unbounded. It is thus possible for the output signal of the flipflop to violate the input constraints of other devices and, by so doing, cause inconsistencies. This accounts for the occasional synchronization failures experienced by existing systems.

RESPONSE TIME

When the input constraints of a flipflop are violated, the input event can leave the circuit in a non-stable state and stabilization then occurs as a result of regenerative feedback. The response time depends on the time needed to stabilize and that depends on the state from which stabilization begins. This relationship is developed analytically in Appendix A for an NMOS flipflop constructed from two cross-coupled inverters. If we add some input gating to that circuit and create a latch with data input D and clock input C, then the sketch below illustrates the dependence of response time on input condition. The ordinate t_r is the circuit's response time and the abscissa t_d is the time of occurrence of a data change as measured from the nearest clock event. The normal response time of the circuit is t_{pd} , and interval M is the range of values of t_d for which the response time is greater than normal.



The value of t_r becomes arbitrarily large near the middle of M. In particular, for any proposed bound t_b , we see that there exists an interval within M for which $t_r > t_b$. The probability of exceeding this bound is thus equal to the probability of t_d occurring in that interval. This observation is quantified in the following equations which are based on the stabilization time properties of the model in Appendix A and on a uniform distribution of t_d .

$$P(t_d \in M) = \frac{I_M}{I_C} \quad (1)$$

$$P(t_r > t_b \mid t_d \in M) = \frac{1}{k + (1-k)e^{(t_b - t_{pd})/\tau}} \quad (2)$$

$$P(t_r > t_b) = \frac{I_M}{I_C} \cdot \frac{1}{k + (1-k)e^{(t_b - t_{pd})/\tau}} \quad (3)$$

$$\approx \frac{I_M}{I_C} \cdot \frac{e^{-(t_b - t_{pd})/\tau}}{(1-k)} \quad t_b - t_{pd} \geq 5\tau \quad (4)$$

$$= \frac{T_0}{I_C} \cdot e^{-t_b/\tau} \quad t_b - t_{pd} \geq 5\tau \quad (5)$$

Parameter I_M is the duration of interval M, parameter I_C is the clock period and is assumed to be constant, and t_b is a bound greater than or equal to t_{pd} . Parameters k and τ are circuit parameters as defined in Appendix A, with k being a positive fraction less than 1 and τ being a time constant with values on the order of a few nanoseconds.

Equation 5 is a convenient form because there are only two circuit-dependent parameters, T_0 and τ , and an experimental method for estimating them is given in [9]. Equation 5 is also in agreement with results obtained experimentally and analytically for other technologies, including TTL, ECL, and a tunnel-diode memory element. There is evidence suggesting that it is not a good estimator for values of t_b close to t_{pd} . Equation 3 seems reasonable in that region but this has yet to be verified experimentally.

IMPLICATIONS

Given that Equation 5 is reasonably accurate, two observations can be made. The first is that there is no finite value of t_b for which $P(t_r > t_b) = 0$. This supports the contention that response time is unbounded. The second observation is more complex and requires the following preliminaries.

An error is the occurrence of a response time greater than the response time available in the application. A failure is an inconsistency caused by an error. Failures occur less often than errors because consistent interpretations are possible even if input constraints are violated. The expected number of errors is denoted $E_e(t_a)$ and, as derived in Appendix B, is given by

$$E_e(t_a) = P(t_r > t_a) \cdot \lambda \cdot t \quad (6)$$

where t_a is the available response time, where λ is the average rate of change of the signal at the data input of the flipflop, and where t is the time over which the errors are counted. $E_e(t_a)$ is the least upper bound for the expected number of failures and is thus a good measure of reliability. The expected number of errors decreases rapidly for larger values of t_a . In particular,

$$E_e(t_a + x) \approx E_e(t_a) \cdot 10^{-x/2.3\tau} \quad (7)$$

The values of τ that have been measured for TTL, ECL, and NMOS circuits are close to 2 nanoseconds. This means that the expected number of errors is reduced by a factor of about 150 each time t_a is increased by 10 nanoseconds. Extremely low error rates can thus be obtained for t_a in the tens of nanoseconds. For example, the box below gives the parameters for a reasonably busy TTL application. The expected number of errors is 1.24 every 10 years. For all practical purposes, the design does not fail.

Parameters

Application	Circuit (TTL)	Design
$I_C = 100 \text{ nsec}$ $\lambda = 10^5 \text{ changes/sec}$	$\tau = 1.8 \text{ nsec}$ $T_0 = 10^{3.07} \text{ nsec}$	$t_a = 60 \text{ nsec}$

This allows us to make our second observation. Even though there is no absolute bound, there are bounds that are adequate from an engineering standpoint. These engineering bounds differ from absolute bounds in that they depend on application parameters as well as on circuit parameters. They must therefore be determined on a case by case basis.

BOUND BASED DESIGNS

The simplest and most economical technique for coping with the absence of an absolute bound is to use an engineering bound. The reliability of this approach depends on the statistical properties of the input signal, on the sample rate, on the circuit parameters, and on the available response time. While the first of these is not under the designer's control, the others are, and this section discusses some of the ways in which he can exercise this control.

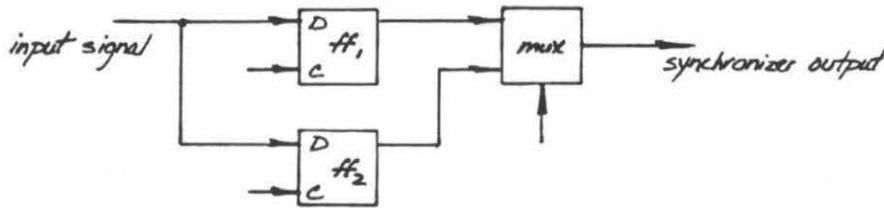
The designer's task in using this approach is to provide enough response time, and a crucial first step is to determine how much is enough. If the statistical properties of the input signal are appropriate, the required time can be estimated by picking an acceptable error rate and solving Equation 6 for t_a . This calculation requires values for λ , T_0 , and τ , where λ is estimated from an analysis of the input environment and where T_0 and τ are estimated for the flipflop circuit that the designer would like to use. Estimating the circuit parameters is a problem because manufacturers do not provide this information. However, representative values for different technologies can be found in the following references: TTL [3, 5], ECL [3, 5], NMOS [9], tunnel diode [1, 5]. If the statistical properties of the input signal do not allow the use of Equation 6, the designer must analyze the properties and develop an equivalent equation. In either case, the derived value is the minimum response time that should be made available. We denote that time by t_b because it is the bound that will be assumed in the design process. The response time that is actually available is denoted t_a , and it is the designer's task to guarantee that $t_a \geq t_b$.

Once a value has been determined for t_b , the designer can make a quick design pass and determine if $t_a \geq t_b$. If it is, fine. If not, more time can be made available by using more than one flipflop in the synchronizer. For example, consider an application with the parameters shown in the box below.

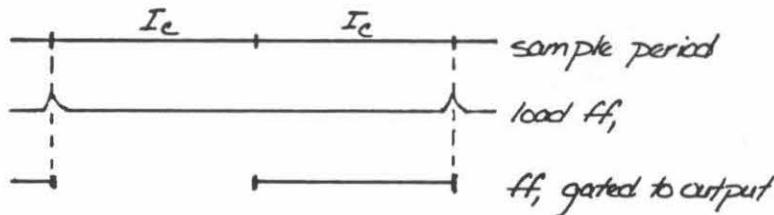
Parameters

Application	Circuit (NMOS)	Design
$I_C = 40 \text{ nsec}$ $\lambda = 10^5 \text{ changes/sec}$	$\tau = 1.6 \text{ nsec}$ $T_0 = 20 \text{ nsec}$	$t_a = 30 \text{ nsec}$

Assuming that Equation 6 applies, a t_a of 30 nanoseconds means that about 31 errors per day can be expected. Since this is too high an error rate, we increase the available response time by using the design shown below.



In this design, the flipflops take turns receiving samples so that, even though the input signal is being sampled every 40 nsec, the clock period for a given flipflop is 80 nsec. The multiplexor gates the data from the flipflops onto a single output path in such a way that the data from a given flipflop is on the path during the sample period preceding the reloading of the flipflop. This is illustrated in the timing diagram below.



The available response time is now the original 30 nsec, plus 40 nsec due to halving the loading rate, minus 10 nsec because of propagation delay through the multiplexor. This is more than adequate since $t_a = 60$ nsec means about 7 errors every 12 thousand years. However, if it were not, more flipflops would be used. The table below shows how the available time increases with the number of flipflops.

<u>Number of Flipflops</u>	<u>Available Response Time (nsec)</u>
1	30 (given)
2	$30 + 1 \cdot 40 - 10 = 60$
3	$30 + 2 \cdot 40 - 10 = 100$
4	$30 + 3 \cdot 40 - 10 = 140$

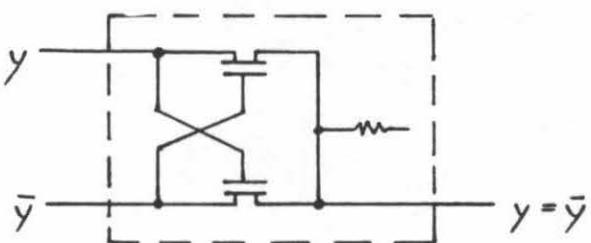
The designer can thus arrange any amount of response time that is called for. There are situations, however, in which t_b must be kept small. The reason is that t_b directly affects the time that it takes for the system

to detect an input change, and performance considerations can require that the detection time be as small as possible. There is only one way that the value of t_b can be reduced without sacrificing reliability. This is to use a flipflop circuit with a smaller value of T_0 and τ . Of the two parameters, τ has the most significant effect, and reducing τ by a factor will reduce the value of t_b by about the same factor. The analysis in [9] indicates that both parameters scale linearly with feature size in NMOS circuits. The VLSI logic designer can thus control the parameters to the extent that manufacturing constraints will allow. In systems constructed of discrete components, extremely small values of T_0 and τ can be obtained by using flipflops constructed of tunnel diodes.

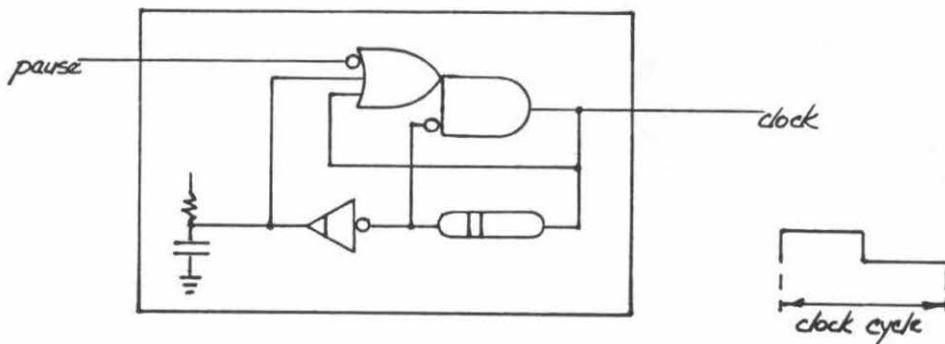
AN ADAPTIVE DESIGN STRATEGY

The design strategy presented in this section differs from that of the preceding section in that reliability is not based on engineering bounds. This scheme monitors the flipflop and, in the case of response times that are long enough to cause errors, it suspends system operation until the response is completed. There are thus no errors. The basic components of this scheme are a flipflop circuit whose response state (stable or unstable) can be monitored, a circuit to do the monitoring, and a clock circuit whose operation can be suspended between clock cycles. The latter is necessary because the monitor suspends system operation by suspending the operation of the clock circuit. This freezes the state of the entire system and guarantees that the flipflop data will not be used until it is stable and logically defined.

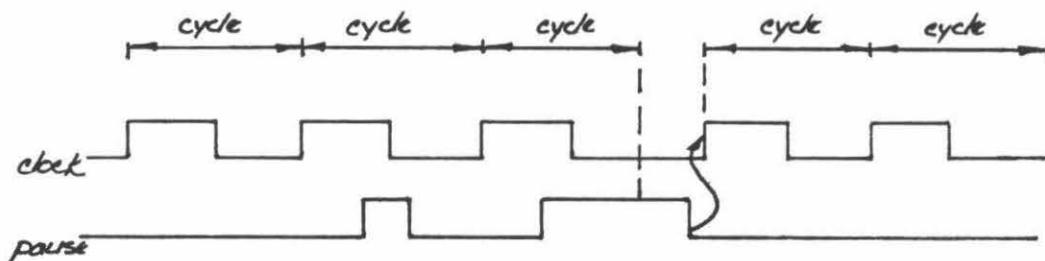
The flipflop circuit used in this design must have an output behavior during a response episode that is uniquely different from its behavior between episodes. The flipflop in Appendix A is such a circuit. During a response episode, the circuit is unstable and its complementary outputs are anti-complementary, that is, they have similar values. The signals can be fairly static, hovering near the middle of the HIGH and LOW logic bands, or they can oscillate, but whatever their behavior happens to be, they will have nearly equal values for the entire episode. This means that the existence of the unstable state can be detected with a difference comparator whose output is a logic 1 if and only if its input signals differ by less than some appropriate amount. The exclusive NOR circuit shown below is just such a detector. Its output is HIGH as long as the input signals differ by less than the threshold voltage of the input gates. The circuit designer sets this threshold at a value that will distinguish the stable and unstable states.



The clock circuit used in this design strategy must be capable of having its operation suspended between clock cycles. The logic diagram for such a circuit is shown below, along with the output waveform for a clock cycle. The waveform can be used as the master clock signal in systems using edge-triggered logic. For systems using pulse-triggered logic, pulses can be generated by adding appropriate output logic.



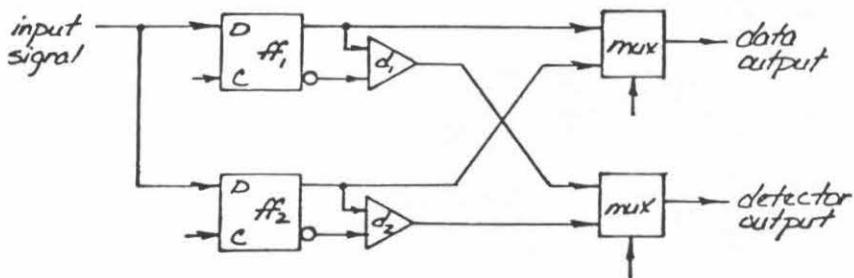
This circuit honors a pause request that is present at the end of a cycle by postponing the next cycle for as long as the request is present. When the request disappears, the next cycle begins immediately. This is illustrated in the diagram below. Note that postponement does not occur if a pause request disappears before the end of a cycle. This is convenient because it means that the detector can request a pause as soon as a response episode begins. If a pause is not actually needed, the request will drop before the end of the cycle and there will be no effect on clock operation.



Since flipflop response time has no absolute bound, it is possible for the operation of the clock to be suspended for an unacceptably long time. To protect against such an occurrence, the clock circuit contains a time-out delay which will automatically restart it if a pause becomes excessive. The length of the time-out delay is controlled by the RC network and is set appropriate to the application.

Proper operation of the clock circuit requires that a pause request not be raised too close to the beginning or end of a clock cycle. This is not a problem in practice because the time at which a pause request is raised is completely predictable. There is no timing constraint on the dropping of a request.

The frequency with which pauses occur and the duration of the pauses are of obvious concern since they affect the performance of the system as a whole. These measures are dependent on the statistical properties of the input signal, on the sample rate, on the circuit parameters, and on the available response time. As in the case of the bound-based strategy, all but the first are under the designers control. Furthermore, since the conditions that lead to an error in the bound-based strategy are the same conditions that lead to a pause in the adaptive strategy, the techniques for controlling error rate can also be used to control pause rate. For example, the multi-flipflop scheme can be used as shown below.



Each flipflop has its own detector d_i and the detector outputs are multiplexed in exactly the same manner as the flipflop outputs. Since the detector for a given flipflop will not be gated through the multiplexor until the sample period preceding the re-loading of the flipflop, a pause will not occur unless instability lasts for at least 2 sample periods. The extension of this scheme to N flipflops is obvious, and a response episode would have to last N sample periods in order to cause the clock to pause. The designer can also affect the pause rate by his choice of flipflop. Circuits with smaller values of τ and T_0 will have shorter response times and will therefore cause fewer pauses. The shorter response times also mean shorter pauses.

Thus, the designer can arrange any amount of response time that is called for. In the case of a uniform input distribution, this time can be estimated by picking an acceptable pause rate and solving Equation 6 for t_a . (The pause rate replaces the error rate in this calculation.) The distribution of pause durations for this case is essentially independent of the available response time. Given that the available time is at least $t_{pd} + 5\tau$, the average duration will be τ , and 98% of the durations will be less than 5τ . Because the pauses tend to be so short, many applications will be able to tolerate a modest pause rate, and as a result, the available response time

can be less than would be required in a bound-based design. This is an advantage in applications where the detection delay must be small.

An important aspect of the adaptive scheme is its ability to resolve the situation where two interacting systems are operating at about the same frequency and phase, and as a consequence, the signals sent between them violate the flipflop constraints with great regularity. In a bound-based design, the error rate would be inordinately high, but in an adaptive design, pauses would perturb the relative operating phase and would tend to disrupt the unfavorable situation, reducing the likelihood of an inordinately high pause rate.

DISCUSSION

Future VLSI designs may use multiple independent clocks that give rise to opportunities for synchronizer failures and ensuing system failures. These failures will be characterized by transient and elusive symptoms that compound the problems of circuit design and verification. Thus careful specification and design must precede the use of a strategy based on bounding the number of synchronization errors. Knowledge of both circuit dependent and application dependent parameters is required for the success of this strategy. Unfortunately, published circuit data are fragmentary, test procedures are unstandardized, manufacturer's specifications are non-existent, and the application dependent parameters may be inaccessible.

One approach to this difficult situation is to design synchronizers with such a large margin of safety that sizable errors in estimating application and circuit parameters can be tolerated. This is practical since the probability of synchronizer failure is reduced by about two orders of magnitude for each 10 nsec added to the allowed flipflop response time.

Thus, a pragmatic solution exists to the synchronizing problem whenever estimates of circuit and application parameters are available and appropriate. There are, however, important circumstances in which a stochastic model of the input to the synchronizer may be inappropriate. Consider, for example, the composition of large systems through the interconnection of modular subsystems chosen from a set of basic module types. This powerful and attractive approach to VLSI design leads to a class of systems with great diversity in the detailed structure of intermodule interaction. Increased care is required in module design to assure that all legitimate compositions of the basic module types will operate correctly. Synchronization is a particularly vexing problem in this context if each module is independently clocked. No a priori estimate of application parameters is available to the module designer, and even if it were, inter-module interactions may not be well modeled by a stochastic process.

Multiple independently clocked subsystems within a single VLSI chip present a particularly interesting and relevant example of this modeling difficulty. If several local clocks are fabricated on the same chip, they

are likely to operate with very nearly the same period unless special precautions are taken. Although most of the time for most systems, operation would be trouble-free, a particular set of processes operating in subsystems driven by a particular set of clocks could lead to an essentially deterministic pattern of transitions that would violate the flipflops input constraints. Because of uniformity in fabrication, in temperature coefficient, and in sensitivity to external fields, response times of synchronizers in such a VLSI system might be consistently long, and the opportunities for system failure might be increased even beyond that experienced in current experiments designed to measure synchronizer failures.

Under these circumstances, an adaptive synchronization strategy appears to be preferable to a probabilistic one. Synchronization is then error free even though the total time to complete a computation may be increased slightly.

CONCLUSION

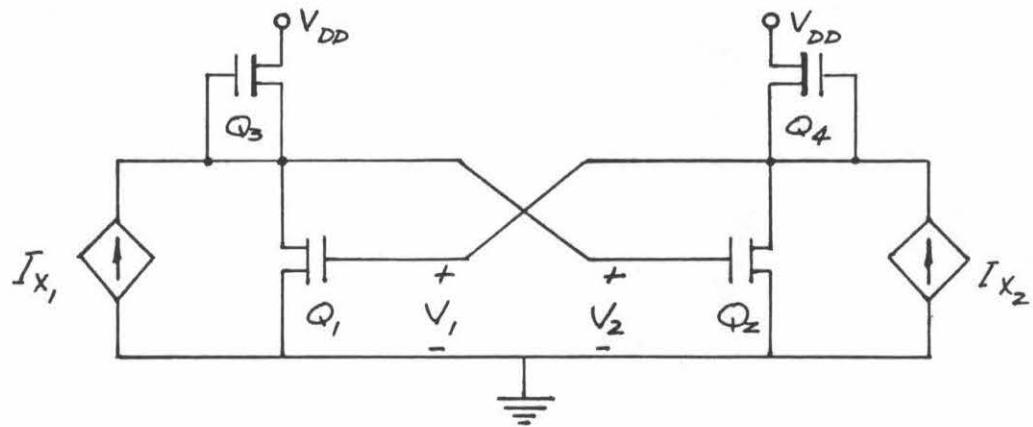
The synchronization problem has been described and synchronizer strategies presented that exemplify the trade-offs possible between failure probability, sample rate, detection time and computation time. The combination of a flipflop and an instability detector, called an indicating flipflop by Kinnemann [5], has been known for more than a decade [7, 10]. Until recently the use of an indicating flipflop has been limited to asynchronous systems, but its incorporation in an adaptive synchronization strategy [11] makes possible error-free synchronization at the expense of variability in the clock period. This strategy and the more familiar probabilistic strategy provide the designer with alternatives for the management of the synchronizer problem. Thus, designers who continue to ignore the problem cannot use the excuse that the problem is fundamentally insoluble. Only design and implementation costs stand in the way of reliable synchronizers. With achievement of the anticipated economies of VLSI, we believe these costs are much less than the verification and maintenance costs associated with poorly designed synchronizers.

ACKNOWLEDGEMENTS

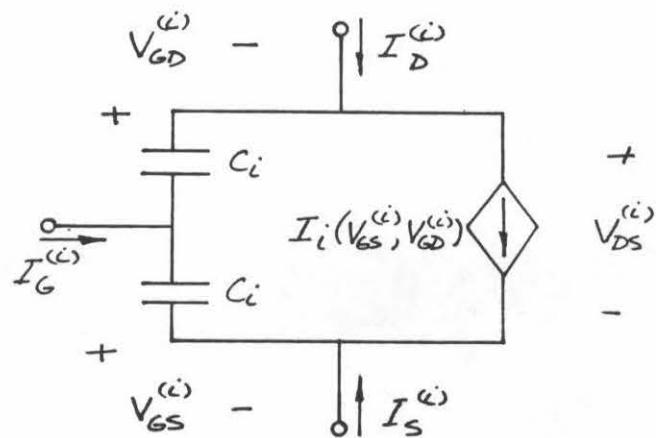
A number of the ideas in this paper were suggested or independently developed by others. Chuck Seitz of Cal Tech built a pausable clock for an Evans and Sutherland graphics system marketed in the early 1970s. Ivan Sutherland designed the MOS instability detector in 1976. Pechoucek independently reported coupling an indicating flipflop and a pausable clock in a paper in 1976. Tom Chaney, Fred Rosenberger and Charlie Molnar have all contributed ideas to this paper through many discussions. We wish to acknowledge here our thanks for their help.

Appendix A

The simple MOS flipflop shown below exhibits the essential features of synchronizer behavior.



Each MOS transistor is modeled by the following equivalent circuit.



The two capacitors C_i split the gate capacitance and include other parasitic capacitance between gate and drain and between gate and source. The current source represents the static drain characteristics expressed as a function of $V_{GD}^{(i)}$ and $V_{GS}^{(i)}$ instead of the more usual, but less symmetric form in terms of $V_{GS}^{(i)}$ and $V_{DS}^{(i)}$,

$$I_i(V_{GS}^{(i)}, V_{GD}^{(i)}) = \begin{cases} \frac{\epsilon \mu_e W_i}{2DL_i} \left[(V_{GS}^{(i)} - V_{TH}^{(i)})^2 - (V_{GD}^{(i)} - V_{TH}^{(i)})^2 \right] ; V_{GS}^{(i)}, V_{GD}^{(i)} > V_{TH}^{(i)} \\ \frac{\epsilon \mu_e W_i}{2DL_i} \left[(V_{GS}^{(i)} - V_{TH}^{(i)})^2 \right] ; V_{GS}^{(i)} > V_{TH}^{(i)}, V_{GD}^{(i)} < V_{TH}^{(i)} \\ \frac{\epsilon \mu_e W_i}{2DL_i} \left[-(V_{GD}^{(i)} - V_{TH}^{(i)})^2 \right] ; V_{GS}^{(i)} < V_{TH}^{(i)}, V_{GD}^{(i)} > V_{TH}^{(i)} \end{cases}$$

(see for example [12]). In these equations ϵ is the oxide region permativity, μ_e is the channel mobility, W_i is the width of the i^{th} transistor gate, D is the oxide depth, L_i is the length of the i^{th} transistor gate and $V_{TH}^{(i)}$ is the threshold voltage for the i^{th} transistor. Each of the voltages within the circuit can be expressed in terms of the node voltages V_1 , V_2 and V_{DD} . Node equations can be written at the drains for Q_1 and Q_2 which include the two current sources I_{x_1} and I_{x_2} . These sources represent the effects of input gating circuitry and are non-zero whenever a change of state is initiated for the flipflop.

$$2C_2(\dot{V}_2 - \dot{V}_1) - (C_1 + C_4)\dot{V}_1 - I_2(V_2, V_2 - V_1) + I_4(0, V_1 - V_{DD}) + I_{x_2} = 0 \quad (\text{A-1})$$

$$2C_2(\dot{V}_1 - \dot{V}_2) - (C_2 + C_3)\dot{V}_2 - I_1(V_1, V_1 - V_2) + I_3(0, V_2 - V_{DD}) + I_{x_1} = 0 \quad (\text{A-2})$$

We assume symmetry such that $C_1 = C_2$ and $C_3 = C_4$. Furthermore, the dimensions of Q_1 and Q_2 are the same as are those for Q_3 and Q_4 . Thus $I_1(\cdot, \cdot)$ is the same function as $I_2(\cdot, \cdot)$. A similar equivalence holds for $I_3(\cdot, \cdot)$ and $I_4(\cdot, \cdot)$. These simplifications lead to the pair of equations

$$2C_1(\dot{V}_2 - \dot{V}_1) - (C_1 + C_3)\dot{V}_1 - I_1(V_2, V_2 - V_1) + I_3(0, V_1 - V_{DD}) + I_{x_2} = 0 \quad (A-3)$$

$$2C_1(\dot{V}_1 - \dot{V}_2) - (C_1 + C_3)\dot{V}_2 - I_1(V_1, V_1 - V_2) + I_3(0, V_2 - V_{DD}) + I_{x_1} = 0 \quad (A-4)$$

Next we observe that the metastable point occurs for $\dot{V}_1 = \dot{V}_2 = 0$ and vanishing inputs with solution of both (A-3) and (A-4) given by

$$I_1(V_0, 0) = I_3(0, V_0 - V_{DD}) \quad (A-5)$$

such that both drains have the same voltage V_0 . The value of V_0 depends on $V_{TH}^{(1)}$, $V_{TH}^{(2)}$ and the fraction $\beta = \frac{W_1}{L_1} \frac{L_3}{W_3}$. The symmetry evidenced in (A-5) is also present in (A-3) and (A-4) allowing us to limit our consideration to trajectories in the plane (V_1, V_2) for $V_1 > V_2$. We assume throughout that V_1 and V_2 are non-negative.

The trajectories in (V_1, V_2) of interest start near (V_0, V_0) since otherwise no metastable behavior would occur. These trajectories will end at (V_H, V_L) where these two voltages are solutions of

$$I_1(V_L, V_L - V_H) = I_3(0, V_H - V_{DD}) \quad (A-6)$$

$$I_1(V_H, V_H - V_L) = I_3(0, V_L - V_{DD}) \quad (A-7)$$

These trajectories lie near the line $V_1 + V_2 = 2V_0$. This is particularly true for the case $2V_0 = V_{DD}$ and $V_H + V_L = V_{DD}$, not unlikely conditions. Using these approximations and solving for the difference mode voltage $V_D = V_1 - V_2$ we get

$$(5C_1 + C_3)\dot{V}_D = \frac{\epsilon \mu e_i W}{2DL_i} \left\{ V_D V_{DD} - V_D^2 - \frac{1}{\beta} V_D V_{DD} \right\} + I_{x_1} - I_{x_2} \quad (A-8)$$

where the further approximations $V_{TH}^{(1)} = 0$ and $V_{TH}^{(3)} = V_{DD}$ have been made. This equation is to be solved for $t > t_{pd}$ for which $I_{x_1}, I_{x_2} = 0$. To simplify matters normalize the voltage by V_{DD} and collect terms

$$\tau \dot{v} = v - \frac{\beta}{\beta-1} v^2 \quad (A-9)$$

where $\tau = \frac{(5C_1 + C_3)2DL_1\beta}{\epsilon\mu_e W_1 V_{DD}(\beta-1)}$ and $v = \frac{V_D}{V_{DD}}$. (A-9) has the solution

$$v = \frac{1}{\frac{1}{v_H} + \left(\frac{1}{v_0} - \frac{1}{v_H} \right) e^{-(t-t_{pd})/\tau}} ; \quad t \geq t_{pd} \quad (A-10)$$

where we note that

$$v \Big|_{t=t_{pd}} = v_0 \quad (A-11)$$

$$v \Big|_{t=\infty} = v_H = \frac{\beta-1}{\beta} \quad (A-12)$$

so that v_0 and v_H are the initial normalized difference voltage at $t=t_{pd}$ and the final voltage, respectively.

The response time t_r can now be found in terms of the stabilization threshold v_S for the normalized difference voltage merely by substituting $v=v_S$ and $t=t_r$ in (A-10). Solving for the initial voltage v_0 required to produce a particular response time t_r yields

$$\frac{v_0}{v_S} = \frac{1}{k + (1-k)e^{(t_r-t_{pd})/\tau}} ; \quad t_r \geq t_{pd} \quad (A-13)$$

where the abbreviation $k = \frac{v_S}{v_H}$ has been used. Define the window for marginal triggering to be the input conditions that lead to $|v_0| < v_S$ or equivalently $t_r > t_{pd}$. For input edges different by $t_d \in M$ we assume these conditions on v_0 hold and furthermore assume that a uniform distribution for t_d leads to a

uniform distribution for v_0 . These assumptions yield the following result for an arbitrary bound t_b .

$$\begin{aligned} P(t_r > t_b) | t_d \in M &= P(t_r > t_b | |v_0| < v_s) \\ &= P(|v_0| < v_B | |v_0| < v_s) = \frac{v_B}{v_s} \end{aligned} \quad (A-14)$$

where v_B is the initial voltage that causes the flipflop to stabilize at precisely the bound t_b . Substituting (A-13) with $t_r = t_b$ and $v_0 = v_B$ into (A-14) we obtain Equation 2 in the text.

Appendix B

Assume the number of logic signal transitions for $t \geq 0$ can be described as a counting process $\{N(t), t \geq 0\}$ having stationary, independent increments with unit jumps, namely, a Poisson process,

$$P(N(t)=n) = e^{\lambda t} \frac{(\lambda t)^n}{n!} \quad (B-1)$$

$$E[n] = \lambda t$$

Random selection with probability p from this process gives a new counting process $\{M(t), t \geq 0\}$ (see [13]), also Poisson, where

$$P(M(t)=m) = e^{-\mu t} \frac{(\mu t)^m}{m!} \quad (B-2)$$

$$E[m] = \mu t = p\lambda t.$$

REFERENCES

1. Couranz, G. R. An Analysis of Binary Circuits Under Marginal Triggering Conditions, Computer Systems Laboratory, Washington University, St. Louis, Missouri, Technical Report #15, November 1969. Also see Couranz, G. R. and Wann, D. F., "Theoretical and Experimental Behavior of Synchronizers Operating in the Metastable Region", IEEE Trans. on Computers, C-24, 1975.
2. Chaney, T. J. and Molnar, C. E., "Anomalous Behavior of Synchronizer and Arbiter Circuits", IEEE Trans. on Computers, C-22:421-422, April 1973.
3. Chaney, T. J., The Synchronizer 'Glitch' Problem, Computer Systems Laboratory, Washington University, St. Louis, Missouri, Technical Report #47, February 1974.
4. Wann, D. F., Molnar, C. E., Chaney, T. J. and Hurtado, M., "A Fundamental Problem Associated with the Physical Realization of Certain Classes of Petri-Nets", Conf. on Petri-Nets and Related Methods, MIT, Cambridge, Massachusetts, S. S. Patil - Editor, July 1975, to be published. Copies available as Technical Memorandum #215, Computer Systems Laboratory, Washington University, 724 South Euclid Avenue, St. Louis, Missouri 63110.
5. Kinniment, D. J., Woods, J. V., "Synchronization and Arbitration Circuits in Digital Systems", Proc. of the IEEE, (England), Vol. 123, No. 10:961-66, October 1976.
6. Marino, L. R., "The Effect of Asynchronous Inputs on Sequential Network Reliability", IEEE Trans. on Computers, C-26, No. 11:1082-90, November 1977.
7. Littlefield, W. M. and Chaney, T. J., "The Glitch Phenomenon", Computer Systems Laboratory, Washington University, St. Louis, Missouri, Technical Memorandum #10, December 1966.
8. Hurtado, M., Dynamic Structure and Performance of Asymptotically Bistable Systems, D. Sc. Dissertation, Department of Electrical Engineering, Washington University, St. Louis, Missouri, 1975. Also see Hurtado, M. and Elliot, D. L., "Ambiguous Behavior of Logic Bistable Systems", Proceedings Thirteenth Annual Allerton Conference on Circuit and System Theory, University of Illinois at Urbana-Champaign, 1975.
9. Chaney, T. J. and Rosenberger, F. U., "Characterization and Scaling of MOS Flip Flop Performance in Synchronization Applications", Proceedings of Conference on Very Large Scale Integration Architecture, Design and Fabrication, Pasadena, California, January 22-24, 1979.

10. Adams, R. L., Castaldo, D. R. and Kurz, G. W., "Real-time Detection of Latch Resolution Using Threshold Means", U. S. Patent 3,515,998, June 1970.
11. Pechoucek, M., "Anomalous Response Times of Input Synchronizers", IEEE Trans. on Computers, C-25, No. 2:133-139, February 1976.
12. Gray, P. E., and Searle, Electronic Principles, Wiley, 1968.
13. Parzen, E., Stochastic Processes, Holden-Day, 1962.

THE TRIMOSBUS

by

Ivan E. Sutherland¹, Charles E. Molnar²
Robert F. Sproull³ and J. Craig Mudge^{1, 4}

¹California Institute of Technology
Pasadena, CA 91125

²Washington University
St. Louis, MO 63110

³Carnegie-Mellon University
Pittsburgh, PA 15213

⁴Digital Equipment Corporation
Maynard, MA 01754

The research leading to this paper was supported in part by the Defense Advanced Research Projects Agency, contract no. N00039-77-C-0185; in part by the NIH Division of Research Resources, grant no. RR00396; and in part by Digital Equipment Corporation.

ABSTRACT

This paper describes a family of communication buses that permit individual senders to communicate with an arbitrary number of receivers and to wait for the last receiver to respond. TRI in the name signifies the use of three wires for sequencing. The bus is speed-independent in that no assumptions about the relative or absolute speed with which bus participants respond to bus signals are required to ensure proper sequencing of bus operations.

Data are passed by two separate mechanisms. First, during normal bus operation a number of parallel data wires are used to transmit individual characters or numbers. The MOS part of the name refers to the fact that the high input impedance of MOS circuits permits us to use these data wires themselves as storage nodes. Second, for debugging, testing, and error recovery a slower serial data path is provided.

This paper also describes a TRIMOSBUS message protocol. The protocol uses sequences of bus cycles to transmit messages of arbitrary length. The unique sender can be selected either on a message by message basis by arbitration, or within a message by mutual consent. We plan to use the TRIMOSBUS and this message protocol in a variety of system designs at Caltech, Carnegie-Mellon University and Washington University.

SECTION I: INTRODUCTION

In this paper, we discuss a bus design intended for communication among several bus participants. The design permits any participant to send information to one or more other participants at once and to delay subsequent communications until the last participant has signaled receipt of the data. The TRIMOSBUS design differs from most asynchronous buses, such as the Digital Equipment Corporation UNIBUS, mainly because of its one-to-many communication capability. Any of the receivers in a TRIMOSBUS may arbitrarily delay the completion of any communication. In bus designs such as the UNIBUS, a single receiver is designated for each transmission, and although other receivers may observe the transmission, they may not arbitrarily delay its completion and thus cannot reliably extract data from it in the absence of a speed specification.

We distinguish three levels of specification in a TRIMOSBUS design. 1) The TRIBUS is a bus sequencing scheme which uses three sequencing wires to permit individual senders to communicate with multiple receivers and wait for the last of them to respond. 2) The TRIMOSBUS augments this basic sequencing scheme with two data transmission mechanisms well suited to implementation in MOS technology. 3) A TRIMOSBUS MESSAGE PROTOCOL accommodates a variety of system communication needs. The reader is invited to imagine alternatives at each level of specification. The designs we have chosen favor the properties of MOS circuits.

First Level of Specification: SEQUENCING

The TRIBUS level of specification deals mainly with the use of three sequencing wires. In transmission of a single datum, only two of the three wires convey signals; on successive transmissions, successive pairs of wires are used in rotating order. By providing three sequencing wires instead of the single clock wire used in synchronous buses, or the two wires typically used in point-to-point asynchronous buses, we are able to detect reliably the completion signal from the slowest of multiple bus receivers.

The TRIBUS minimizes the number of sequential transitions required to transmit each datum. At most, three transition times are required, one to establish the correct data value on the data lines, one to signal that the data are valid, and one to signal that all receivers have received the data. It is difficult to conceive of a bus design which uses fewer transitions per communication while still providing for acknowledgement from multiple receivers. Other asynchronous one-to-many buses, such

as IEEE 488 [1], use more signaling transitions to transfer a single datum. The use of a minimum number of transitions is especially advantageous in MOS circuitry because its relatively low output current makes off-chip transitions slow.

The TRIMOSBUS design relies upon the low output current property of MOS circuitry to avoid transmission line problems. Because of this low output current and because of the relatively small physical size of systems built with highly integrated MOS components, signal transition times can be kept long compared to transmission line delays. In effect, one can treat each signaling wire as an equipotential node rather than as a transmission line. In a transmission line environment, the task of communicating from one sender to many receivers is made more difficult by the need to accommodate transmission delays and to avoid reflections in the transmission line. An equipotential environment is free of these difficulties.

How long can a TRIMOSBUS be? In typical MOS circuits today, off-chip transition times are measured in tens of nanoseconds, and so propagation delays of about a nanosecond, which correspond to bus lengths of about 20cm, should generally satisfy the equipotential assumption. Buses with greater physical extent must be operated more slowly. It appears that the equipotential assumption may be satisfied for any bus length if the ratio of the bus drive current to its capacitance per unit length is sufficiently low and the interconnecting path is lossless. For high resistivity interconnects, such as diffused or polycrystalline silicon paths, this simple analysis does not suffice; at the higher circuit densities anticipated in the future the scaling of both distances and interconnect properties needs further examination.

The reliable one-to-many communication of the TRIBUS presents several opportunities to ease debugging and testing. For example, the bus can be "single-stepped" simply by controlling one receiver's response with a single-step switch. Because the bus requires responses from all receivers to operate, it will be stalled until the single-step receiver is released.

Second Level of Specification: DATA FLOW

An important attribute of MOS circuits is the high impedance of inputs, and of outputs that are disabled. This permits data wires to be used easily as temporary storage nodes. In the TRIMOSBUS design, a sender drives the data wires to the desired value, and senses the voltage on these wires. When the data wires reach the correct voltage, the sender turns off the driving current source before signaling that the data are ready.

Thus, a sender disconnects as soon as a data value has been delivered to the bus, rather than having to wait for the slowest receiver to capture it, and the bus itself provides a level of buffering for each transmission.

The TRIMOSBUS design includes several features designed to facilitate error detection, debugging and testing. For error recovery and testing, the TRIMOSBUS design includes a separate serial communication line as part of the bus. We intend that each bus participant include a serial shift register debugging mechanism which can, upon command, report the content of key state registers in the bus participant. Control of this shift register is obtained by highly redundant coding on multiple bus wires, so that it will operate in spite of severe malfunction of the bus. Serial communication is used to minimize the number of pins required for this function.

Third Level of Specification: MESSAGES

We have designed a simple message protocol for use in systems of integrated circuits. This message protocol provides for messages that consist of one or more consecutive bus cycles. The last cycle in each message is marked so that all bus participants may easily distinguish the end of each message and, hence, the beginning of the next message. Our protocol allows senders to transfer the right to use the bus as a part of the message format; arbitration between contending senders need be used at most once per message.

The protocol is defined in terms of some simple codes used to herald various types of messages. The coding space available for such heralds is not nearly filled. We hope to add message types to provide for a rich variety of messages in systems involving multiple processors and memories. For example, messages to report system status and to assist in debugging and testing could be included.

SECTION II: SEQUENCING

Outline of Operation

The TRIMOSBUS contains two kinds of interconnection paths which are terminated differently: three sequencing wires and an arbitrary number of data wires, as shown in Figure 1. The three sequencing wires are used in a wired-or configuration. Each of them is terminated with a pullup resistor, as shown in Figure 1, which, in the absence of drive from any of the bus participants, will cause it to assume a logically inactive state. In N-channel MOS and TTL circuitry, this inactive state for a wired-or signal is the HIGH state. The bus participants can clamp one or more of these sequencing wires to the active state, the LOW state in TTL or N-channel MOS designs.

The data wires are terminated with negative resistance to a voltage source at the switching threshold so that they will remain in either the HIGH or LOW state for an unlimited time after they are driven to that state and the drive is removed. The negative resistance termination is weak enough that it can easily be overpowered by any of the drivers of the bus, but strong enough to maintain logically defined signal levels in spite of noise pickup and charge leakage. Stray capacitance between the bus wires and ground is, of course, an additional stabilizing influence.

A communication on the bus requires three successive transitions on bus wires. In the first transition, a single sender, selected from several which may be ready to send by an arbitration mechanism to be described later, places its data onto the data wires of the bus. This requires one transition time unless all data wires are already in the proper state. The sender senses the state of the data wires and removes its drive when it observes them to be in the correct state. Because of the negative resistance termination, the data wires will retain their state indefinitely long even after the drive is removed.

After removing drive from the data wires, the sender generates the second transition, a sequencing signal indicating that valid data are present on the data wires. Sequencing signals appear on the three sequencing wires in rotation. Before the DATA VALID transition, one of the sequencing wires is clamped in the LOW state and two are unclamped and in the HIGH state, as shown in Figure 2a. This is one of three equivalent idle bus states as shown in Figure 2a, f and h. The sender generates its DATA VALID signal by clamping the "next" signaling wire in rotation order to the LOW state, as shown in Figure 2b. After the DATA VALID transition, two of the three sequencing wires are in the active (LOW) state and one is

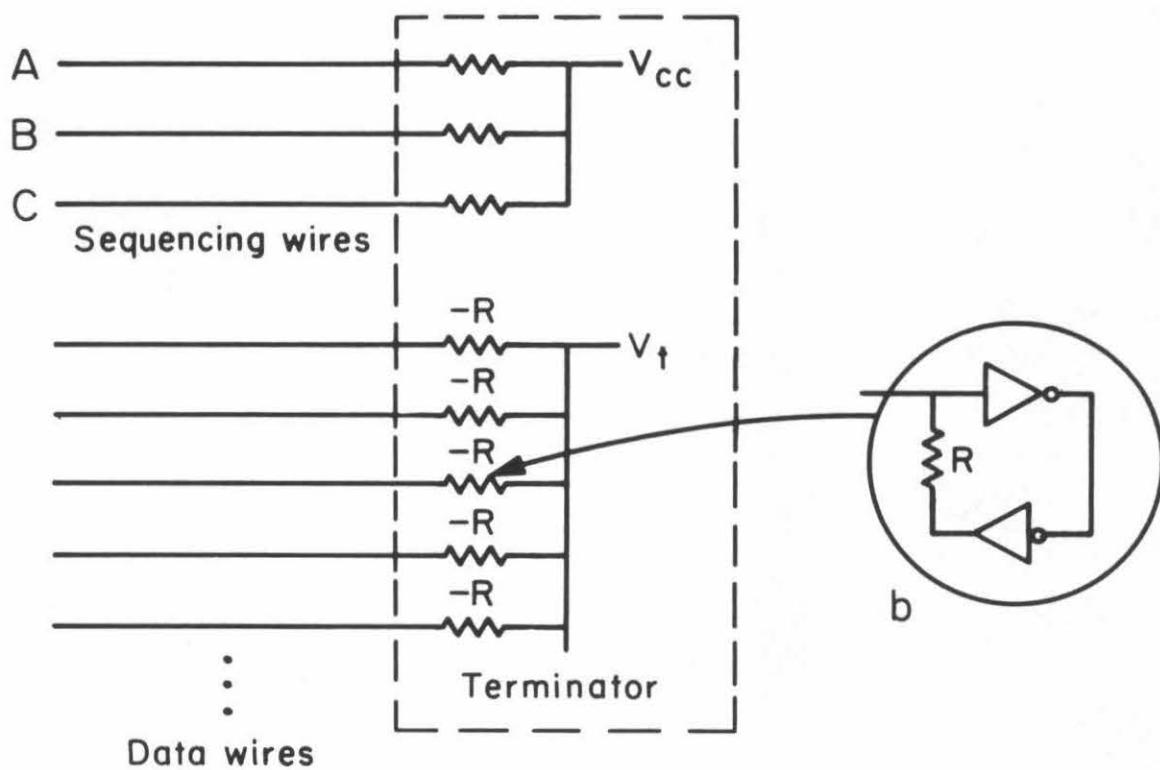
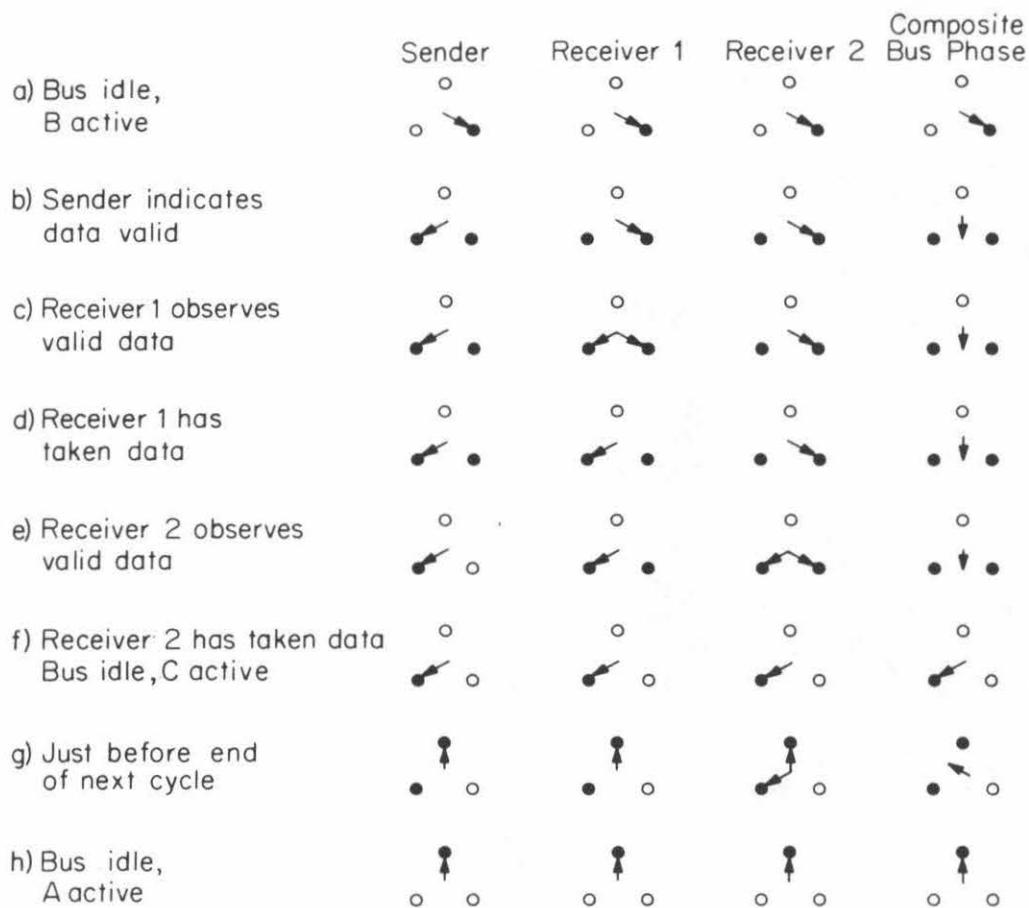


Figure 1: The TRIMOSBUS uses two sorts of wires. Three sequencing wires are terminated to V_{CC} to permit wired-or operation. An arbitrary number of data wires is provided, each terminated with a negative resistance to the inverter threshold voltage V_t . This negative resistance termination allows the wires to be used reliably as storage nodes. Inset: one implementation for the negative resistance termination.



- Inactive signaling wire ("High" for NMOS or TTL)
- Active signaling wire ("Low" for NMOS or TTL)
- ↗ Wire clamped to active state

Figure 2: Three-phase bus sequencing is used to indicate valid data and to wait for all receivers to acknowledge receipt of the data. Because sequencing wires are terminated to Vcc, any unclamped sequencing wires assume the "inactive" state. The three idle states of the bus are illustrated in cases a, f and h. We will use the term "bus phase" to distinguish the sequencing states.

inactive (HIGH), which indicates that valid data are available on the bus, but have not yet been accepted by all receivers.

Receivers recognize that data are valid when they observe the DATA VALID transition. While the bus was idle, each receiver was clamping the single active sequencing wire in the active (LOW) state. Upon observing the DATA VALID transition, each receiver must clamp the next sequencing wire in the active (LOW) state as well, as shown in Figure 2c and 2e. Then, when it has satisfactorily received the data, each receiver unclamps the originally clamped sequencing wire, as shown in Figure 2d and 2f. When the final receiver unclamps the originally clamped sequencing wire, the resistive termination will cause the third transition, DATA ACCEPTED, by pulling it into the inactive (HIGH) state, as shown in Figure 2f, thus signaling to the sender and to all other bus participants that all receivers have satisfactorily received the data and that the bus is free to begin the next cycle. Note that the bus has now sequenced to its next idle state. Figure 3 shows how data transitions interleave with sequencing transitions in several successive bus cycles.

Three sequencing wires are required to provide unambiguous indication that all receivers have successfully received the data. To achieve speed-independence, one must ensure that a fast subsequent sender cannot cause confusion in a slow receiver. Thus, one cannot permit two consecutive signaling transitions on the same sequencing wire, because a slow receiver might not observe them. It follows that if only two wires were used, successive transitions would have to occur on alternate wires. This is not feasible in a bus with more than two participants for the following reason. In order to achieve a one-sender multiple-receiver bus with a minimum number of sequential signaling transitions, a transition that signals data validity must be given by a single sender, which may be any of the bus participants; a second transition that signals data acceptance requires agreement by all bus participants. Hence, the former must be accomplished by a "wired-or" connection and the latter by a "wired-and". If these transitions are to alternate and yet not occur consecutively on the same wire, there is no way to use only two wires, since this would require that two adjacent transitions on the same wire be both "wired-or" or both "wired-and".

If a third wire is available, it is possible to satisfy both the condition that two consecutive transitions must not occur on the same wire, and that alternate transitions on the same wire be an alternation of "or" transitions and "and" transitions. Such a three-wire design does require that the functional interpretation of a given signal value on a given

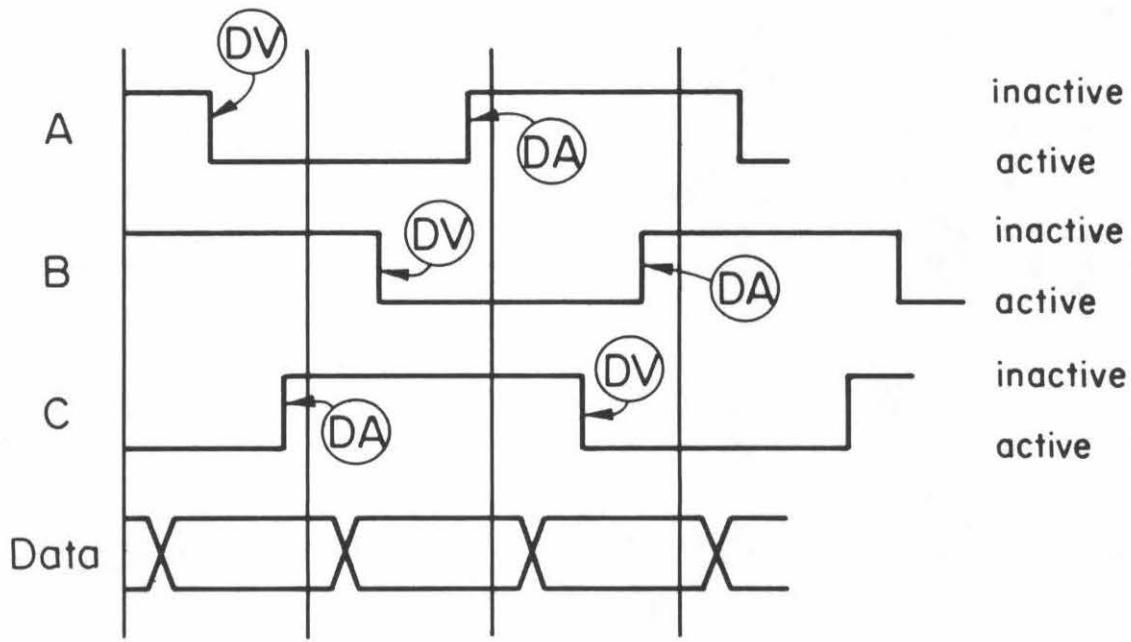


Figure 3: Each bus cycle requires three transitions. The first drives data wires to the correct state. The second indicates data validity (DV) by clamping a sequencing wire. The third indicates data acceptance (DA) by all receivers when a sequencing wire becomes inactive.

wires not be the same for all bus cycles. This property was previously encountered in a two-wire bus design that was considered and rejected for use in Restructured Macromodules [2].

The reader is no doubt concerned, as are we, at the extravagance of three sequencing wires, but they are required for reliable speed-independent operation with more than two participants. As will be seen below, we have shared their cost by using them to provide master clear and maintenance functions as well as sequencing.

The TRIMOSBUS requires that data values be stored on the data pathways as charge on their shunt capacitance. This is a consequence of the sparseness of the sequencing signals that are exchanged on the signalling wires. If timing specifications are to be avoided, a sender must be sure that bus data are valid before sending a DATA VALID transition. Removal of drive from data paths is a more complex matter. If the sender waits to remove drive until after the DATA ACCEPTED signal, it is certain that data validity has been maintained until all receivers have taken the data. However, if the sender is slow in then removing drive from the data paths, it is possible that the next sender will drive the data paths while they are still being driven by the previous sender. This condition appears undesirable and may produce erroneous data values if the next sender removes its data path drive while the previous sender is still driving.

A second alternative is for the sender to remove data path drive after sending the DATA VALID signal without waiting for DATA ACCEPTED. Once again, if the sender is slow in removing drive and all of the receivers and the next sender are fast, overlapping of data path drive by two senders is possible, unless the sender also acts as a receiver and ensures that the DATA ACCEPTED signal on the bus does not occur until after data path drive is removed. In this latter case, however, there is no way to ensure that data path drive will be maintained until all receivers have taken the data. Thus, if conflicting drive of data paths by two senders is to be avoided, while at the same time maintaining assurance that data remain valid on the data path until taken by all receivers, it is essential that data validity, once attained, be maintained by the data path even after data path drive is removed by the sender.

One way of doing this is to equip the bus pathways themselves with storage ability by terminating them with a negative resistance in such a way as to make a bistable circuit. For MOS technology, in which gate inputs and gate outputs in the "OFF" condition can be made to have very high impedance values, it is also possible to achieve "dynamic" storage without such negative resistance terminators. Although this violates our

goal of speed independence, since some upper limit, determined by leakage currents, is then placed on the length of time that data values on the bus path can safely be assumed to remain valid after drive is removed, this may still be a practically useful approach.

A Design for the Sequencer

TRIMOSBUS sequencing has a three-fold circular symmetry; the sequencing wires are used in succession, repeating their function every three cycles as shown in Figure 2. This symmetry permits one to think of the transitions on the sequencing wires as changes in the "phase" of the bus sequence. The three-fold symmetry of sequencing suggests that the sequencing control should also have three-fold circular symmetry. Indeed, we have found that a simple control mechanism can be implemented using tri-flops, the tri-stable analog of the bi-stable flip-flop. This section of the paper will describe a simple control circuit for TRIMOSBUS senders and receivers which we have built and tested. The control is shown in Figure 4.

The control for each sender and receiver contains a tri-stable circuit to keep a record of the most recent bus phase as shown in Figure 2a, 2f and 2h. The control detects phase changes in the bus by comparing the bus phase to the phase of its tri-flop. The control causes changes on the sequencing wires by advancing the phase of its tri-flop.

Senders and receivers can detect transitions on the sequencing wires with simple circularly symmetric logic functions which relate the actual phase of the bus to the recorded phase. Thus, for example, "bus ahead" might be used to describe a logic function $A*Sc+B*Sa+C*Sb$, in which A, B, and C represent the active states of the three sequencing wires and Sa, Sb, and Sc represent the corresponding internal states of the tri-flop. By thinking of the relationship between the bus phase and the tri-flop phase in terms of such circularly symmetric functions, one is led quickly to simple bus control designs.

In N-MOS or TTL logic, the wired-or arrangement for the bus wires is a LOW-active configuration. Thus when the three sequencing wires are in the idle state, one will be LOW and active and the other two HIGH and inactive. If we call the three wires A, B, and C and assume that sequencing is in that order, then if the bus is idle with B LOW and A and C HIGH, new bus activity will be signaled by C going LOW. The circularly symmetric function "LOW AHEAD" implies that the bus wire ahead of the internal state of the control has changed to the LOW or active state indicating DATA VALID. The circularly symmetric

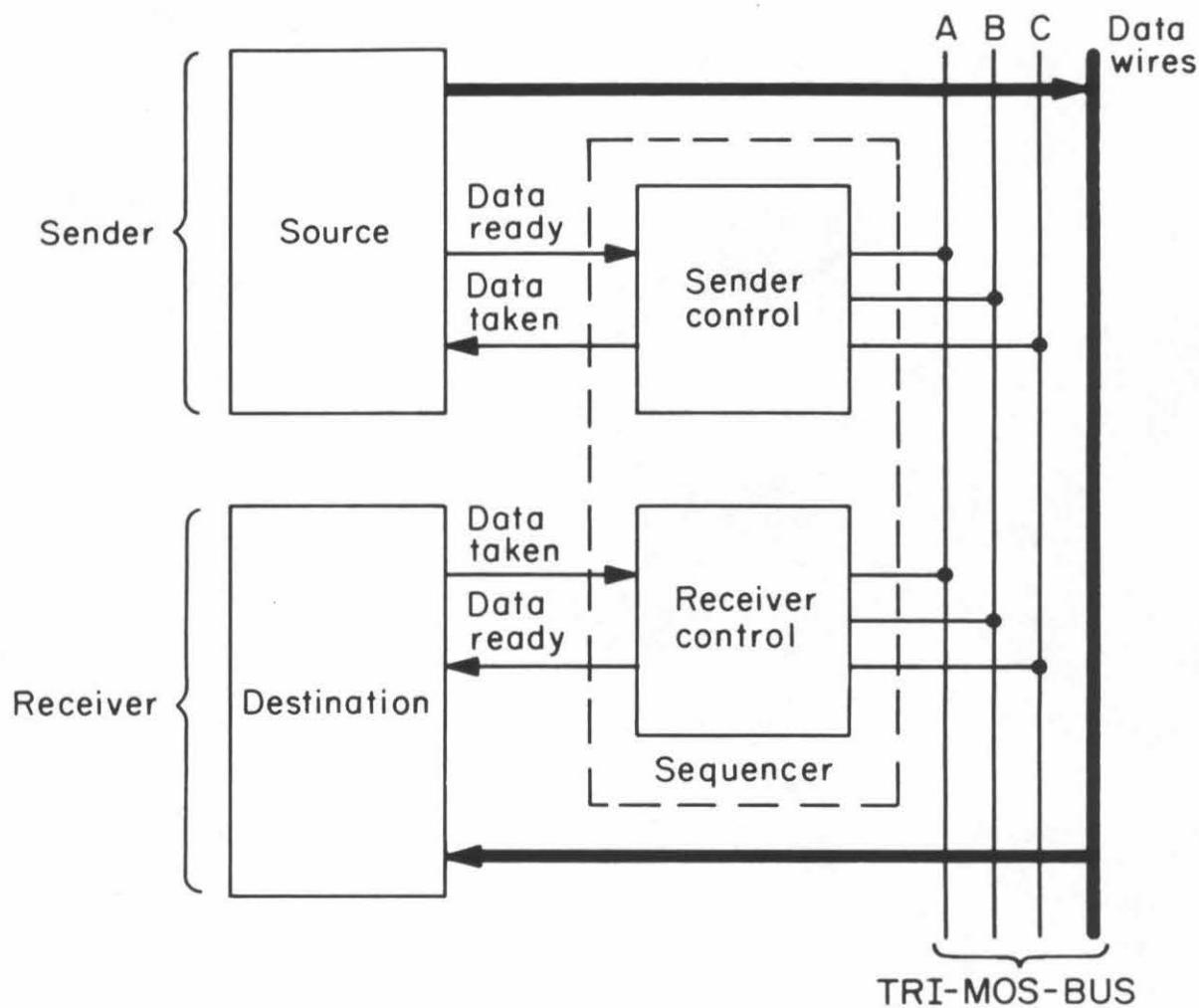


Figure 4: Schematic diagram of a bus participant. The source and destination will drive and sense the data wires respectively. The sequencer converts between TRIMOSBUS sequencing signals and conventional two-wire handshaking signals.

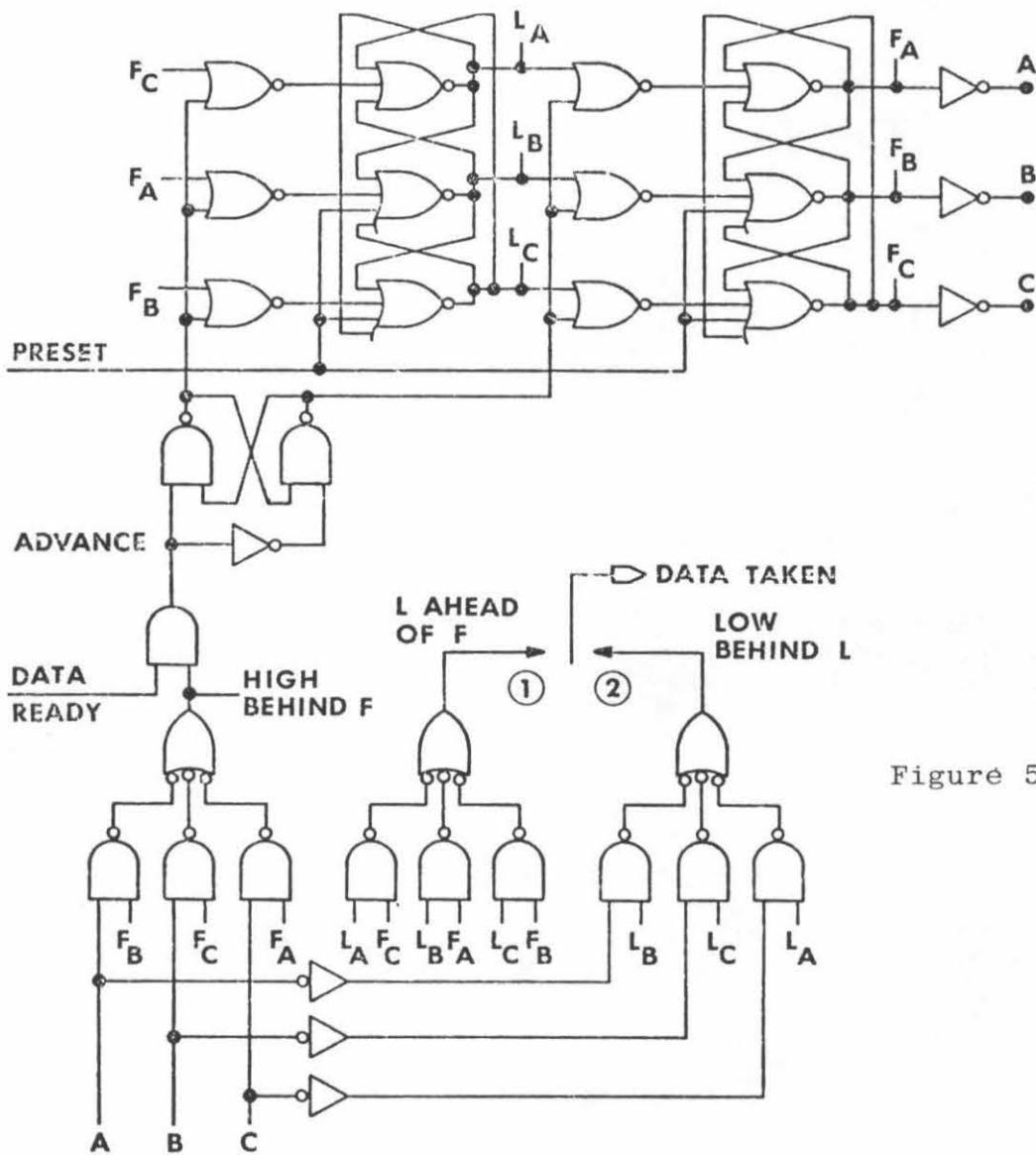


Figure 5a

Figure 5: Logic diagram for sender (5a) and receiver (5b) of experimental TRIMOSBUS control for use as shown in Figures 4 and 6. The DATA READY signal in the sender circuit may be derived from source (1) or source (2), corresponding to the signaling interpretations shown in Figures 6a and 6b. The receiver circuit can be seen to be very similar to that for the sender.

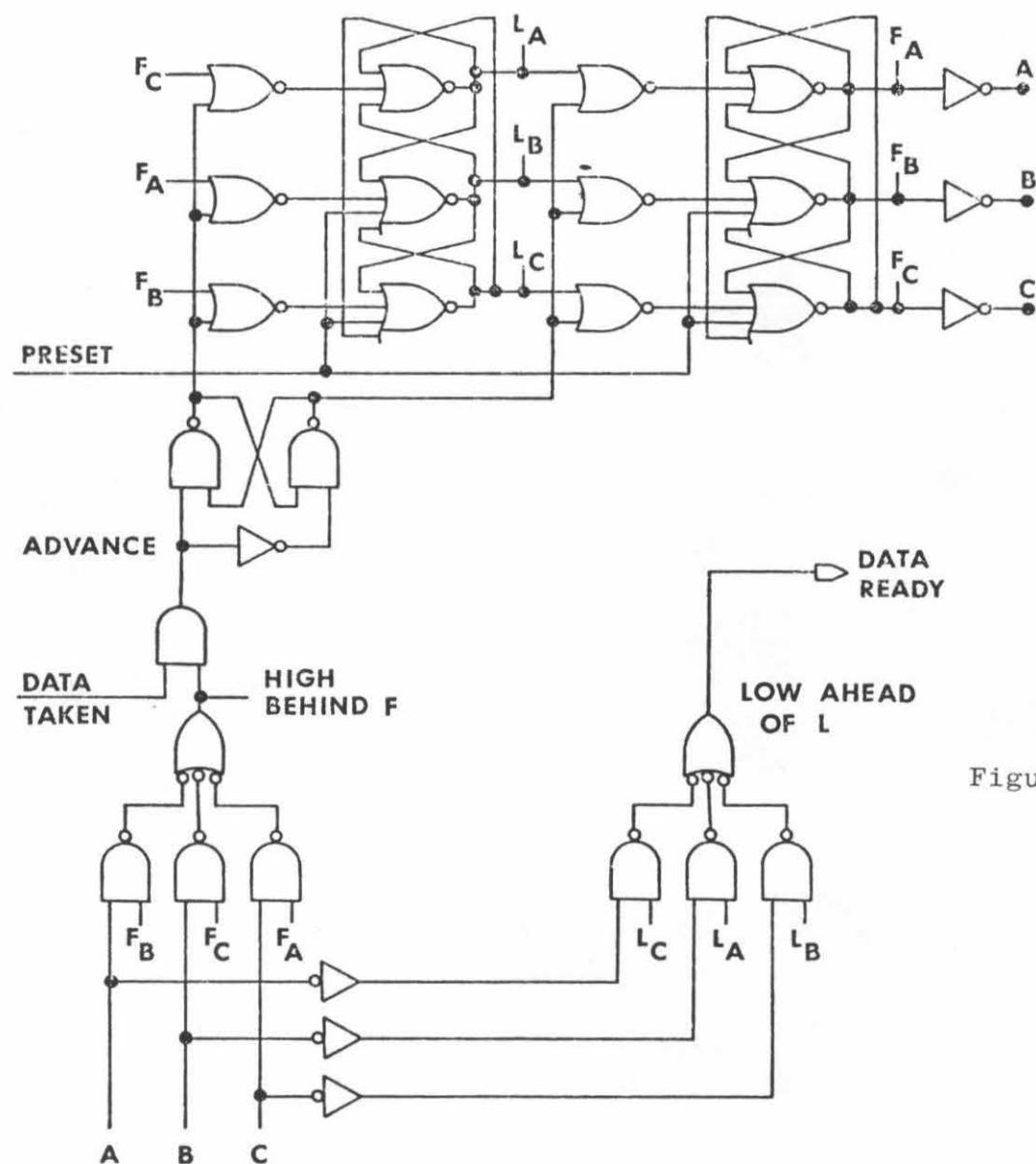


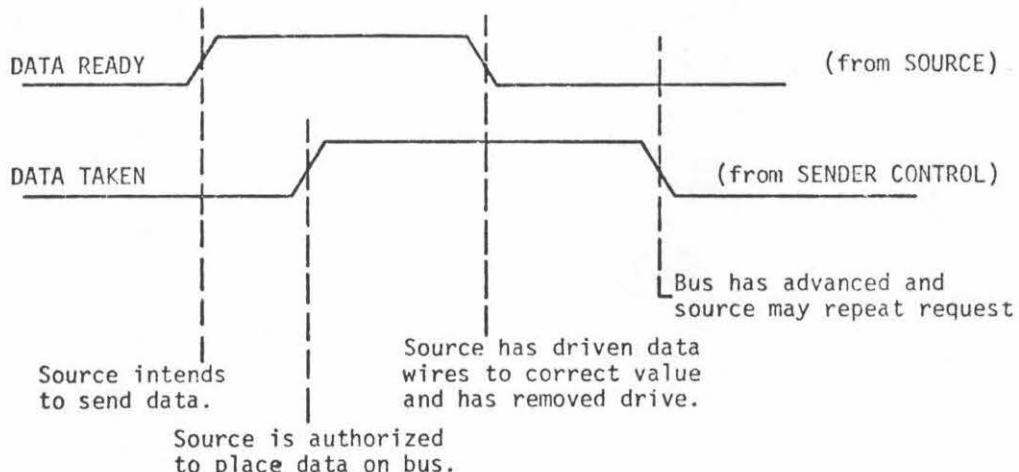
Figure 5b

function "HIGH BEHIND" implies that the bus wire behind the current state has returned to the high or inactive state, indicating that the last receiver has accepted the data from the previous cycle. LOW AHEAD thus heralds new activity, while HIGH BEHIND signals completion of former activity. Because three sequencing wires are used, the HIGH BEHIND and LOW AHEAD conditions are not mutually exclusive and may be observed at the same time. Because three rather than two signalling wires are used, these two conditions can nevertheless be unambiguously signalled even if they should appear to overlap in time.

A simple experimental bus control circuit is shown in Figure 5. This control circuit is intended to convert two-wire four-phase handshake signals as shown in Figure 6 to the signalling conventions of the TRIBUS. This design was intended as an experiment to demonstrate a simple bus example with one sender and two receivers. It does not include provision for multiple senders or for arbitration among competing senders, although any number of receivers up to the limits of distance and circuit constraints may be used. The control circuits for sender (5a) and for receiver (5b) are quite similar in that both use a dual-rank tristable circuit that is connected as a ring counter. When the ADVANCE line is asserted, the contents of the following (F) tri-flop are copied into the leading (L) tri-flop. When the ADVANCE line is not asserted, the contents of the leading tri-flop are copied into the following tri-flop. The outputs of the following tri-flop provide drive to the three transistors that can clamp bus signalling paths A, B, and C to the low state. Since only one of the three outputs of a tri-flop is high when the tri-flop is in a stable condition, the sender and receiver controls shown here always clamp exactly one bus signalling wire while in a given stable state.

The combinational circuits at the bottom of Figures 5a and 5b generate the ADVANCE signal and signals to the SOURCE component of the sender and the DESTINATION component of the receiver. In the sender, a SOURCE that has data to send asserts DATA READY. When the bus becomes inactive, as indicated by the HIGH BEHIND F condition, ADVANCE is asserted and the leading tri-flop is loaded with the shifted contents of the following tri-flop. This does not change the clamping of the bus signalling paths, but the change in the leading tri-flop causes the condition L AHEAD OF F to be satisfied. This generates the assertion of DATA TAKEN to the SOURCE, signaling that the bus data paths are free and the sender control is primed to advance the bus. Upon receipt of this signal, the SOURCE places its data values on the bus data wires. When they have reached a valid condition, the SOURCE removes its data path drive and then deasserts DATA

a) SENDER OPTION 1



b) SENDER OPTION 2

Identical to 1, except that the last transition indicates that all receivers have taken the data and source may repeat request.

c) RECEIVER

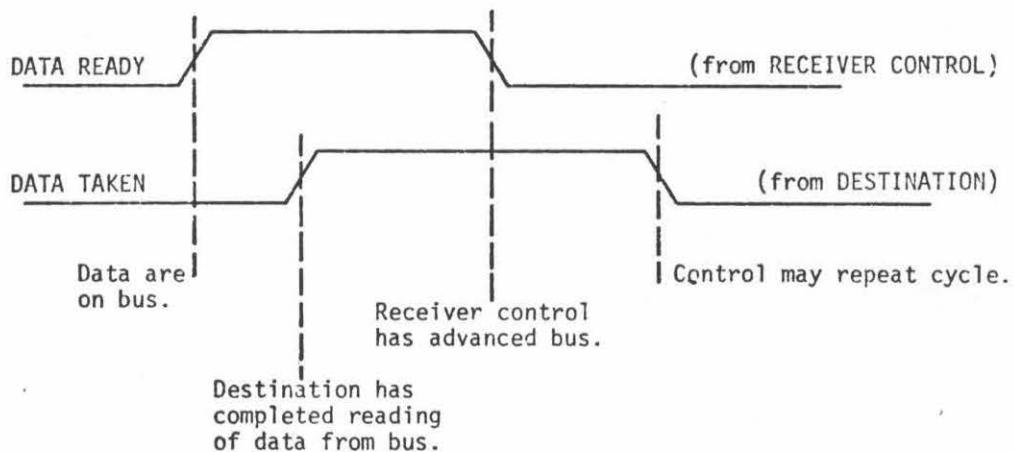


Figure 6: Two-wire handshaking signals used by the sequencer. Two variations of the sender control are shown. The first allows the data wires to provide a single level of buffering between the sender and the receivers.

READY. This in turn causes the deassertion of ADVANCE, which causes the following tri-flop to be loaded with the contents of the leading tri-flop. This advances the bus by clamping the next signaling wire in rotation.

The advancing of the following tri-flop also causes the HIGH BEHIND F and L AHEAD OF F conditions to be no longer satisfied. This removes the assertion of DATA TAKEN to the SOURCE, enabling the SOURCE to begin preparation of its next request to send on the bus. The HIGH BEHIND F condition does not hold until the bus completes the cycle and all receivers have taken the data from the bus data wires. Response of the sender control circuit to the next assertion of DATA READY by the SOURCE cannot begin until HIGH BEHIND F holds.

Alternatively, the DATA TAKEN signal to the SOURCE may be generated from the LOW BEHIND L condition, as indicated by option 2 in Figures 5a and 6. In this case, the assertion of DATA TAKEN follows the loading of the leading tri-flop as before; the deassertion of DATA TAKEN is however held up until the LOW BEHIND L condition is removed by the signal from all bus receivers that they have taken the data.

The receiver control operates in a similar manner. The advancing of the bus by a sender causes the LOW AHEAD OF L condition to hold, generating a DATA READY signal to the DESTINATION. The DATA TAKEN signal that follows causes ADVANCE to be asserted, which advances the leading tri-flop and removes the LOW AHEAD OF L condition. This in turn removes DATA READY. Following the receipt of the deassertion of DATA READY, and after it has completed the taking of data from the data paths, the DESTINATION deasserts DATA TAKEN, thereby allowing the deassertion of ADVANCE, which allows the bus trailing signaling wire to be unclamped and a DATA ACCEPTED signal to be generated on the bus signaling wires when the last receiver has accepted the data.

An experimental TRIBUS has been built and tested using a TTL implementation of the circuits of Figure 5a and 5b. It was operated successfully over a wide variety of internal delay conditions. We have observed timing asymmetry introduced by loading one sequencing wire heavily with shunt capacitance; correct sequencing was maintained in spite of loading.

In a detailed and complete design for the bus control circuit, careful attention must be given to avoid race conditions within the circuit. We do not view such a requirement as compromising our intention that the bus design be speed independent; any circuit design that satisfies the signaling sequence conditions at the sequencer terminal is acceptable.

SECTION III: DATA FLOW

The data communication mechanism in the TRIMOSBUS uses the bus wires themselves as a storage register. The negative resistance termination on the bus wires has already been described in Figure 1. One can, of course, use as many data wires as one chooses.

Speed independence in the presence of variations in the electrical characteristics of individual data wires and drivers can be guaranteed by source checking. The source detects the state of the data wires and signals DATA VALID only when it senses all data wires to be correct. Source checking eliminates the data transition time entirely if data values for two successive bus cycles are the same. Source checking also can detect certain transmission errors, such as those caused by short circuits on the data wires, leaving the bus stopped in the offending state.

Extenders

The equipotential assumption may limit the practical length of a TRIMOSBUS to dimensions of a few feet in MOS and no more than a few inches in faster technologies. Point-to-point extension of the bus, however, is relatively straightforward. The idea is that two TRIMOSBUSS remote from each other might be connected by a cable of arbitrary length and delay. A suitable controller connects each end of the cable to its TRIMOSBUS. These controllers serve as senders or receivers on their respective TRIMOSBUSS and communicate with each other through the cable with a traditional point-to-point asynchronous signaling scheme.

Two interconnections possible in such a network of TRIMOSBUSS are: 1) all of the buses in the network are forced to operate in rigid sequence, since each of the point-to-point controllers will delay completion of any transmission until its point-to-point companion reports completion; 2) each extender may provide a store-and-forward mechanism, allowing the separate TRIMOSBUSS to sequence concurrently. In the latter case, of course, one must provide means to avoid choking the extenders with data and thus causing some form of deadlock.

Arbitration

Although there can be any number of receivers, the TRIMOSBUS design assumes a unique sender for each bus cycle. The task of selecting a single sender from many contenders that may asynchronously request permission to send on the bus is called arbitration. Arbitration must be attended to carefully, since there are a number of pitfalls which can cause low probability system failures that are very difficult to find and correct [3].

In a practical TRIMOSBUS design there are two ways in which sender selection may be done. First, arbitration will not be required if the system design calls for fully sequential operation. This is the case, for example, in systems in which a single CPU sends read requests to multiple memories. Each memory must receive all memory requests and address values, so that it may decide whether or not the request is addressed to it, but the TRIMOSBUS design requires an address assignment scheme that ensures that only a single memory will respond.

On the other hand, there are systems that must have many independent senders: for example, systems with collections of processors operating together, or with channel controllers which communicate independently with memory. In this case asynchronous arbitration using any of a variety of techniques [4] can be used to select a single unique sender. It remains to be shown here only how to adapt such schemes to the protocol of the TRIMOSBUS.

An important and somewhat subtle question is what is the earliest time that the arbiter may safely signal to the next sender that it is authorized to initiate a message. If a message consists of only a single bus cycle, then the arbiter may not designate the next sender until the arbiter has seen the HIGH BEHIND condition that signals the end of the current bus cycle. This is necessary because otherwise there is no way for the arbiter to be sure that the next sender has already seen the beginning of the current cycle (indicated by LOW AHEAD). Thus, if the arbiter sends the designated next sender a signal before seeing the HIGH BEHIND of the current bus cycle, it is possible that the next sender will attempt to initiate a bus cycle concurrently with the current bus cycle. Because the next sender (like all other bus participants) must have recognized the current bus cycle and accepted it before HIGH BEHIND could occur, it is sufficient that the arbiter observe HIGH BEHIND to ensure that the next sender has already recognized the current cycle.

If a message consists of more than one bus cycle, another method will allow the next sender to be designated earlier than the HIGH BEHIND transition of the last cycle of the current

message, thus allowing overlap of data transmission with designation of the next sender. All that is necessary is: 1) that the arbiter observes the HIGH BEHIND condition following the first cycle of the current message before designating the next sender; and 2) that the next sender has a means of identifying the last cycle of the current message. In this way, there can be no ambiguity, since the next sender must recognize the beginning of the current message before it receives from the arbiter the signal designating it as the next sender. This, plus the ability to identify the last cycle of a message, removes all ambiguity.

SECTION IV: DEBUGGING, TESTING AND ERROR CONTROL

The TRIMOSBUS design makes explicit provisions for debugging, testing, and error control. Although a bus terminator is required primarily to provide proper electrical termination for the three sequencing wires and for the data-transmission wires, it is a convenient place to house debugging aids as well. The terminator contains a bus receiver and a short shift register which records a history of recent bus data values. The receiver also provides the ability to "single-step" the bus by refusing to release its clamp on the "previous" sequencing wire until an external signal to the terminator indicates that the bus may proceed. Both the history and single-step functions can be controlled by connecting the terminator to a computer system that provides debugging functions.

The terminator also detects certain types of errors on the bus and reports them to the debugging computer. We have chosen to devote one data wire of the TRIMOSBUS to data parity; the terminator constantly monitors bus cycles to detect and report parity errors. Many kinds of errors on the sequencing wires can also be detected, such as an individual wire going high and then low again without any activity on the other wires. The terminator can also detect prolonged inactivity on the bus and notify the debugging computer that the bus has "timed out". It is necessary to introduce the notion of "time" into the TRIMOSBUS only to detect inactivity; the timeout interval can, however, be made arbitrarily long.

Other bus participants can also help to detect errors. Each one can independently check the parity of the bus, a test which serves to uncover bad sockets or inoperative bus receiver circuits. Also, a participant may discover errors within itself that compromise any further operation. In either case, the error may be reported simply by failing to let the bus sequencing wires advance, thereby causing the terminator to detect a bus timeout. Alternatively, a failed bus participant may remove itself from participation in bus sequencing by ceasing to clamp or drive any wires.

Serial Communication

If the occurrence of an error halts normal bus operation, either because the error has rendered the bus inoperative or because a bus participant has deliberately stalled bus operation, we need to be able to inspect the state of individual system participants by a means independent of normal bus operation. For this purpose, the TRIMOSBUS links all participants

together in a single serial connection shown in Figure 7. This connection is used to shift state information into and out of the participants to which it is connected. A debugging computer can examine this state and modify it if necessary.

The notion of making a chip's state available by a serial connection is not new. IBM has incorporated such an idea into standard integrated circuit design practice [5]. Several manufacturers link printed circuit boards with a shift register to provide a debugging computer access to vital system state [6,7]. What we have done in the TRIMOSBUS design is to define this facility as part of the bus itself.

Although the serial connection itself requires only two additional pins for each bus participant, some mechanism must be provided to sequence the shift registers. Additional controls are also desirable: for example, to read the participant's state into the shift register, or to write the participant's state from the shift register. Separate reading and writing controls greatly simplify testing, as they permit arbitrary values to be inserted in registers and flip-flops that otherwise could not be tested exhaustively. These control functions and the shift register clocking signals are encoded in a highly redundant form on the bus data wires; thus, although the bus may not be fully operational, we assume that most failures will allow it to work well enough to transmit the needed codes. This mechanism, called HHH signaling, is taken up in the next section.

HHH Signaling

When normal bus operation is prevented, it is nonetheless essential to transmit a small number of codes to all system elements. One such code, INIT, is needed to initialize the system and put all receivers in a state in which they are clamping the same sequencing wire. Three additional codes are needed to control the serial line: a command to READ the machine state into the shift register, a command to SHIFT it, and a command to WRITE the machine state from the shift register. These four codes, and possibly others, are transmitted over the data wires by the terminator in a redundant way, so that the failure of any one bus wire or its connectors or receivers will not prevent detection of the code. These wires also carry a "code validity" signal, or "clock", in redundant form. The coding scheme we have chosen requires that the bus have at least nine data wires.

To commandeer the data wires for this debugging function, the terminator drives all three sequencing wires to the inactive

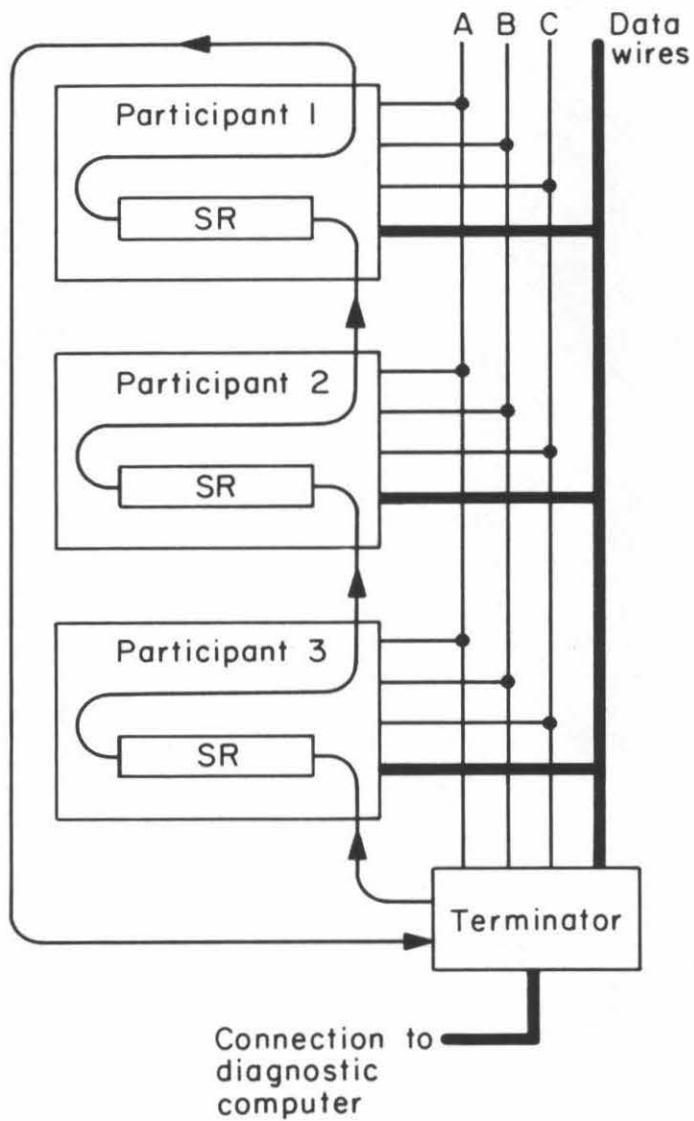


Figure 7: . A serial communication line is used to read and write the state of the bus participants for debugging.

state, HIGH in NMOS implementations. This requires substantial drive capability, because various participants in the system may be clamping one or more of these wires to the active state. However, as soon as a participant detects the "all high" (HHH) condition, it removes all drive on sequencing and data wires. Then it looks for codes and clock on the data wires which will control recovery or debugging.

The HHH signaling mechanism is not speed-independent. No acknowledgement is provided by any of the bus participants. Consequently, information must be transmitted slowly enough to ensure that even the slowest receiver responds properly.

SECTION V: MESSAGES

Although the basic signaling conventions of the TRIMOSBUS are sufficient to transmit information from one bus participant to another, these conventions establish no interpretation for these signals. Communicating among elements in a computer system linked by the TRIMOSBUS requires another and higher level of protocol specification. Our choices for this level of design are motivated by our desire to use a TRIMOSBUS to construct systems containing experimental MOS integrated circuits. Although some of these systems may use the bus in a conventional way to link processors and memories, some may wish to experiment with more exotic communication protocols.

Bus Width

Choosing the width of buses in a computer system is a delicate process, for it inevitably constrains the performance that can be achieved by the communication system. In addition to the three sequencing wires, we have chosen to provide ten wires for transmitting information from the sender to all receivers: eight data wires, a "tag" wire, and a parity wire that is set to guarantee odd parity of all ten wires. This choice is primarily driven by pin limitations: we want a design that allows small integrated circuits to connect easily to the bus.

Although HHH signaling requires a minimum bus width of 9, nothing prevents expanding the bus to arbitrary widths. The only difficulty in such expansion is to devise a scheme that allows chips with differing bus widths to be connected together. The store-and-forward bus extenders described in an earlier section could, for example, be used to link busses of different width.

Messages

Bus participants communicate with one another by sending messages, each of which requires a sequence of consecutive bus cycles. Because these messages may be arbitrarily long, the TRIMOSBUS can be used to transmit objects that exceed eight bits in size. Each of the receivers in the system must decode, or "parse" the messages it receives on the bus. Not all messages will be valuable to a particular bus participant, but it must nevertheless inspect each one to determine its relevance.

Although the message mechanism can be put to several uses in a computer system, it will be illustrated by considering the conventional communication between a processor and its memories. A processor will construct a message that contains all the information required for a memory to "write" a new value at a given address, and transmit the message over the TRIMOSBUS. Of all the system components that decipher the message, only one particular memory should recognize the address as its responsibility, and perform the requested write operation.

The message generated by the processor contains three parts: a "herald" that indicates that the message is a "write" command, an address, and a data value. Generally, a transmission of all of these parts would require more than one bus cycle because more than eight bits are needed. A message to write a 16-bit value in a 24-bit address space would typically use a single bus cycle for the message herald, three for the address, and two for the data value.

Correct operation of the system requires that all system elements parse messages according to the same conventions. This requirement does not impose a rigid structure on messages. Rather, the decoder can be viewed as a finite-state machine that takes as inputs the successive data values transmitted, beginning with the message herald. After one or more cycles, the state machine may determine that the current message is of no interest, and wait for a new message to begin. The state machine may also "accept" the message, and cause its bus participant to take action. This acceptance is not to be confused with acknowledging each bus cycle, which must be done in every case.

Proper parsing of messages requires a synchronizing mechanism to ensure that all receivers begin parsing when a message herald is transmitted. This synchronization is achieved by marking the last bus cycle of a message with the "tag" bit. All bus cycles except the last set the tag wire to zero; the last cycle of a message sets it to one. This synchronizes the receivers in a way that makes it easy to construct variable-length messages.

Transferring Sendership

The TRIMOSBUS message conventions do not require that every bus cycle of a message be transmitted by the same sender. Instead, the initial sender can hand control of the bus to the next sender, which in turn may hand control to a third party,

or back to the original sender, etc. The conventions for transferring sendership must be rigidly enforced, for there must always be exactly one next sender.

The transfer of sendership is best illustrated with a "memory read" operation. The processor transmits a message herald that specifies a "read", followed by the address of the memory location to be read. Then the processor falls silent, and relies on the specific memory element that recognized the address to become the sender, so that it may transmit the data value requested as soon as it is available. The memory tags the last bus cycle, in order to terminate the message, and, by convention, sendership reverts to the processor.

The ability to transfer sendership allows very simple systems to be built that require no bus arbitration mechanism. A single bus participant starts all messages, which may be completed by memories or input/output devices that respond with requested values.

Message Arbitration

If the TRIMOSBUS provides communication among a number of asynchronous processors, an arbitration mechanism is required to decide which may use the bus. The arbitration mechanism operates at the message level, i.e., it determines which bus participant may use the bus to transmit the next message, which may consume several separate bus cycles. This approach allows arbitration decisions to proceed rather slowly compared to the speed of bus transmission without limiting bus performance.

It may be necessary to permit a sender to retain control of the bus in order to transmit several consecutive messages without interruption. For example, if multiple processors share a TRIMOSBUS to communicate with memory, the familiar "test and set" operation might require a read message and a write message to occur without relinquishing the bus. Alternatively, the entire operation might be defined as a single message.

The Message Repertoire

Message protocols in the TRIMOSBUS can be designed to meet special needs faced by particular systems. The examples we have used that illustrate a processor communicating with several memories are only simple cases. Generally, the messages will deal with objects and operations that are implemented in

the chips or boards connected to the bus. Communication with functional units such as floating-point arithmetic modules, or "execution contexts" will lead to more sophisticated message formats than we have illustrated. Object-oriented systems such as Smalltalk [8] or SIMULA [9] may use messages that specify an object name and an operation to perform on that object.

The message protocols we have designed all require each receiver to "associate" on some part of the message to decide whether it must act. The processor-memory example requires each memory to decode a memory address and to decide whether it contains the data associated with that address. More generally, a message contains a "name" of an object on which to operate. The object may be the responsibility of more than one bus participant; a memory cache is a simple example in which a data value is stored both in a cache module and in a primary memory module. As another example, one could imagine an aircraft collision-detection system in which the TRIMOSBUS broadcasts positions of aircraft when they change. One of the participants will use this information to update its understanding of the aircraft's new position. Other bus participants will receive the information and compare it to positions of aircraft for which they are responsible, to see if a collision is possible.

The associative nature of the message-parsing process is more suited to the TRIMOSBUS than are explicit physical origin and destination addresses. Specific addresses would fail to exploit the one-to-many transmission offered by the TRIMOSBUS by requiring an explicit destination address. Association also allows dynamic configuration by altering which bus participants are responsible for which names. Although association may require somewhat more circuitry in a bus participant, the highly integrated circuitry used to implement these modules may render such cost negligible.

It is interesting to note that TRIMOSBUS messages can assume several different roles in conventional computer systems. The bus performs well enough to be used as a processor-memory interconnection. However, it can connect objects of enough processing ability to be used the way computer networks are now. Thus message repertoires may occasionally mix low-level communication protocols such as memory fetches with others that resemble high-level network protocols [10]. As the level of integration of TRIMOSBUS participants increases, message protocols will look less like a processor-memory connections and more like high-level network protocols.

SECTION VI: A FAMILY OF BUS DESIGNS

In the preceding description of the TRIMOSBUS, we have intertwined our discussion of the TRI and MOS aspects of the design. The basic three-wire sequencing mechanism, the TRIBUS, is applicable to a variety of technologies. Indeed, our prototype controller is implemented in TTL. The discussions of the equipotential assumption, arbitration, and bus extenders are likewise associated with the TRIBUS.

The TRIMOSBUS adapts these ideas for MOS implementation. The most important observation is that MOS implementation makes it easy to allow the bus data wires to be storage nodes. Terminating these wires with negative resistance increases noise immunity and reduces power consumption in all chips except the terminator.

Methods for debugging, testing, and recovering from errors are integrated into the TRIMOSBUS design. The integration is desirable in part because components such as the terminator and the data bus wires can be used both for normal bus operation and for the less frequent interventions. The integration is also desirable to encourage a style of system design that recognizes from the outset the need to deal with errors, debugging, and testing. Clearly, a similar philosophy can be implemented in technologies other than MOS.

The TRIBUS and TRIMOSBUS designs, as discussed here, represent "bus families", rather than completely specified buses. Different circuit designs, different bus widths, different communication distances, different arbitration schemes, and different higher level message protocols can be found that are consistent with the basic ideas expressed here.

SECTION VII: CONCLUSIONS

We have described three levels of specification for a one-to-many self-timed communication bus. These levels provide a framework within which one might describe a variety of specific bus systems. Readers are invited to design their own favorite communication system within the framework.

One-to-many communication in systems which contain parts with different and perhaps unknown response times seems entirely feasible, and in retrospect, fairly simple. The sequence in which all participants in such a communication detect the elements of communication must be consistent. This consistency requirement seems to imply a direct relationship between the physical size of such a bus and its speed. We have satisfied this requirement in the TRIMOSBUS design with the "equipotential assumption"; we assume that the rise time of all signals is slow compared to the maximum propagation delay. We speculate that a rigorous proof of the necessary conditions for safe operation may be found.

Although large systems can be built with point-to-point interconnections between separate TRIMOSBUSSs, such systems must either run relatively slowly or be organized so as to localize most communication within the individual TRIMOSBUSSs, and to use the extensions relatively infrequently. Thus, we find that digital systems can obtain high speed performance only by careful system organization. This seems to us to be a system level version of the speed requirement handled in individual TRIMOSBUSSs with the equipotential assumption. As we have expressed elsewhere [11], the difficulties in contemporary system design stem mainly from communication problems and not from logic design issues. The limitations on TRIMOSBUS performance are similarly related to communication.

ACKNOWLEDGEMENTS

The ideas in this paper came from many sources. To the best of our recollection, recognition of the need for three-wire signaling (TRI) originated in the Washington University group, following an unsuccessful attempt to design a two-wire scheme. The idea of using the bus wires themselves for storage (MOS) arose at Caltech from a suggestion by Chuck Seitz. Fred Rosenberger of Washington University made a number of valuable suggestions.

BIBLIOGRAPHY

- 1) Institute of Electrical and Electronics Engineers, "IEEE Standard Digital Interface for Programmable Instrumentation," IEEE 488-1975.
- 2) "Restructured Macromodules," Technical Report 49, Computer Systems Laboratory, Washington University, St. Louis, p. 25, Feb. 13, 1974.
- 3) T. J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," IEEE Trans. TC-22, 4, pp. 421-422, April 1973.
- 4) C. L. Seitz, "System Timing," Chapter 7 in C. A. Mead and L. A. Conway, Introduction to VLSI Systems, Computer Science Dept., Caltech, in manuscript, 1979.
- 5) E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability." Proc. 14th Design Automation Conference, June 20-22, 1977.
- 6) Used by Digital Equipment Corporation in the VAX-11/780 central processor, 1977.
- 7) Used by Evans and Sutherland in CAORF digital image generator, 1975.
- 8) A. Goldberg, A. Kay (eds), "SmallTalk-72 Instruction Manual," Xerox Palo Alto Research Center, SSL76-6, March 1976.
- 9) O. J. Dahl and K. Nygaard, "SIMULA--An ALGOL-Based Simulation Language," Comm. ACM, 9, 9, p. 671, September 1966.
- 10) R. F. Sproull and D. Cohen, "High Level Protocols," IEEE Proc., 66, 11, p. 1371, November 1978.
- 11) I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," Scientific American, September 1977, p. 210.

ABSTRACT

Timing Considerations in Logic Arrays and Their Importance
to Self Timed Digital Circuits

Suhas S. Patil
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract: This paper presents a method for the design of self timed circuits on an integrated circuit that takes advantage of certain temporal constraints that are realizable in logic arrays.

This method of design recognizes two distinct environments for circuits, the local environment and the global environment and further recognizes that the assumptions regarding the temporal characteristics of the system that may be valid in the local environment may not be valid in the global environment.

This paper shows how self timed circuits can be systematically designed on a single chip using assumptions reasonable for components on a single chip.

This paper is based on our work on Structured Logic Arrays (SLAs) and first explains the intricacies of the temporal constraints implemented in the structured array and then shows how one can take advantage of these constraints in the design of self timed circuits. In this structure, for example, it is possible to design an asynchronous sequential state machine with non adjacent transitions without getting into hazardous conditions. What is presented is related to Petri nets and their realization in circuits.

The circuits that are designed using this method are very regular in structure and are efficient in utilization of chip area. Furthermore, fairly large integrated circuits can be designed relatively fast using this method. Examples of some chip designs are presented.

ARCHITECTURE SESSION

Chairperson:

J. Craig Mudge, Digital Equipment Corporation, Visiting
Associate Professor of Computer Science at Caltech

ARCHITECTURE SESSION

Research aimed at discovering computer structures which match VLSI technology is in its infancy. The invited papers on the first day of the conference gave more than adequate motivation for this fertile research area.

The papers in this session address three areas which represent fundamental directions that we expect to be taken.

A. PMS-Level Design

We expect that the majority of computer design projects (although not the majority of computers manufactured) will use processor-memory-switch (PMS) level components. Such single-chip components include central processors, I/O processors, communications controllers, and complete memories.

The economics of the semiconductor industry make it essential that integrated-circuit suppliers produce circuits with a high degree of universality. This is because the learning curve of a manufacturing process causes cost to be inversely proportional to volume, and for a design to be sold in high volume, it must be usable in a large number of applications.

Thus, chips with high degrees of universality have the lowest cost, and systems constructed from such chips are of low cost. Sequin's paper, Single-Chip Computers - The New VLSI Building Blocks, describes X-tree, a binary tree interconnection of PMS-level components.

B. Communication, Not Logic, Represents the True Cost in VLSI

Five of the papers address this fundamental property of the new technology. The papers by Schorr, High-Speed VLSI Machines, and Sites, How to Use 1000 Registers, are concerned with reducing data movement in machines which implement conventional instruction sets.

To fully exploit VLSI technology, ultra-concurrent architectures need to be developed. Three papers address this: Mago describes a machine which implements Backus's applicative language; Browning reports work on a tree machine designed to closely match Simula, an object oriented language; and Davis describes a data flow architecture.

Davis's paper also gives an excellent summary of the design constraint space and supplies useful data on logic-to-pin ratios and gate counts. The paper is unique in this session in that a prototype of his proposed architecture has been constructed.

C. New Analysis Tools Are Needed

Thompson's area-time model of computation is a new piece of complexity theory which concretely reflects an important part of the cost function of VLSI technology.

The paper by Guibas and his co-workers uses two important combinatorial problems to analyze direct implementation of algorithms in VLSI.

A fourth fundamental direction, unfortunately not covered by these papers, is the impact of escalating design time. New intra-chip structures, which exhibit a high degree of regularity (such as the PLA), need to be developed. Hopefully, future conferences can report such advances.

ABSTRACT

VLSI AND HIGH PERFORMANCE COMPUTERS

Herbert Schorr
IBM T. J. Watson Research Center
P.O.Box 218
Yorktown Heights, N.Y.10598

This talk will outline three items concerned with the application of VLSI to high performance computing machines. These are:

1. Direction of VLSI and System Design Variables.
2. Systems Design Options.
3. Technological problems in building a VLSI machine.

Item one will summarize trends in bits per chip for memories and circuits per chip.

Item two will briefly review design options available to large system designs as a result of the forthcoming high degree of integration (i.e., improved machine organization, increased instruction concurrency, new subsystem concurrency).

Item three will outline some key VLSI technology problems relevant to VLSI machine design. Examples are:

Pins, packaging and performance interactions;
switching noise and technology limitations;
design automation problems and requirements.

SINGLE - CHIP COMPUTERS, THE NEW VLSI BUILDING BLOCKS

Carlo H. Séquin

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley CA 94720

ABSTRACT

Current trends in the design of general purpose VLSI chips are analyzed to explore what a truly modular, general-purpose component for digital computing systems might look like in the mid 1980's. It is concluded that such a component would be a complete single-chip computer, in which the hardware for effective interprocessor communication has been designed with the architecture of the overall multiprocessor system in mind. Computation and communication are handled by separate processors in such a manner, that both can be performed simultaneously with full efficiency.

This paper then describes relevant features of **X-TREE**, a research project which addresses the question how the power of VLSI of the next decade can best be used to build general purpose computing systems of arbitrary size. In **X-TREE**, a general VLSI component realizable in the mid 1980's is defined, and its interconnection into a hierarchical tree-structured network is studied. The overall architecture, communications issues and the blockdiagram of the modular component used are discussed.

1. INTRODUCTION

By now it seems to be a well accepted truth that computing demand will always stay ahead of the available computing power. No matter how much computational power can be realized on a single VLSI circuit, there will always be customers that want to have computing power that substantially exceeds the capabilities of a single chip. (Even though the fraction of those customers will decrease dramatically compared to the number of single-chip users.)

If a system is to be extended over several integrated circuits, the question arises how it should be partitioned onto the various chips. This question is crucial since the connections between chips represent an inherent communications bottleneck. The number of interconnection points on a VLSI chip is limited and increases at a much slower rate than the number of gates in the circuit. Furthermore the bandwidth through those pins is limited; with current packaging technology, the physical paths leading from one chip to the next are one or two orders of magnitude longer than on-chip interconnections, and possess correspondingly larger parasitic capacitances.

It is thus important to find the right functional blocks to be packaged onto individual chips. We can state immediately that the partitioning should not cut through the high-bandwidth path between main memory and the actual processing circuits. Thus memory and processing elements should stay tightly coupled on one and the same chip. VLSI technology in the mid 1980's, will make it possible to pack a sophisticated processor and a substantial amount of memory on a single integrated circuit.

2. GENERAL PURPOSE VLSI COMPONENTS

No matter how fast, convenient and sophisticated VLSI design tools will become, standard off-the-shelf parts will always be in strong demand for people who want to put together experimental systems or who want to bring a product to market quickly. They cannot afford to start from scratch and to

determine in all details, how the optimum chip for their purpose could be constructed. For small and medium volume production, it is far more cost effective to pick a component from a limited selection of existing parts. If these parts have already been debugged and if applications manuals are available, the user need not worry about all the critical details of internal timing and interconnections. It normally pays to choose a somewhat overdesigned component, using more circuitry, power or silicon area than absolutely necessary, thus guaranteeing that the requirements of the application will be safely met.

The question thus arises what type of circuits can be placed on the shelf and will prove useful in a large number of applications. Within the frame work of digital computing systems, what kind of chip will take on the former role of NAND gates, flop-flops or registers ? Simple extrapolation of existing trends may not be the right answer for general purpose components with a hundredfold greater functional capability.

Memory chips with four times more storage capacity appear on the market about every three years now. Certainly, a 64k-word by 1-bit chip is a nice part to have, but will a 1M-word by 1-bit chip be equally attractive ? - Or would most customers rather use a 64k-word by 16-bit part ? Also, from the point of view of optimal systems partitioning, discussed in the previous section, a simple memory chip is a very poor component. It has virtually no internal interactions, and every time it is used, information has to go in and out through the package pins. These shortcomings should be remedied by adding processing power onto the memory chip itself.

Looking at the field of random logic, the current trends are more obscure. There is a certain evolution from the NAND gate to more complicated Boolean functions such as multi-bit logic functions, parity checkers, arithmetic functions or priority encoders. However there is a limit to the size of useful logic functions that are commonly used. The trend is to add input or output registers to the logic functions to permit easy assem-

bly of synchronous machines. Here again, we find a mixture of processing and storage elements.

The most complicated parts available today are obviously the single-chip microcomputers with a certain amount of on-chip memory. But can these chips really be regarded as general purpose, modular building blocks in the sense of a NAND gate? While one can prove that all computing systems could be built from nothing but NAND gates, it is rather doubtful that we can interconnect many, say, INTEL 8048 and produce a computing system of superior power! What is missing is an answer to the question how several such components should be interconnected and how they should interact. While successful TTL parts have been designed so that many of them can be combined to form a system with more capabilities than the sum of its parts, the interconnection issue was definitely not a primary concern in the construction of the first-single chip microcomputers. In the design of a truly modular VLSI component for large computing systems, the communications issue can no longer be neglected.

3. INTER-CHIP COMMUNICATION

Standardizing the communication between computers is a considerably more complicated issue than standardizing interconnections between Boolean logic chips. In addition to voltage levels and load considerations, which apply to logic chips, interconnection of computers involves additional issues such as link allocation, timing, message formats, addressing and routing. All these issues have to be addressed in the design of a truly modular general purpose VLSI component. Hardware for effective support of these features has to be incorporated on the chip. The block diagram of a potential general purpose component, consisting of at least a processor, memory, a communication switch and its controller, is shown in Fig.1 using PMS notation [Bell & Newell 1971]. The case for such "Computer Modules" with proper consideration of the communications issues was first made by Bell *et al.* [1973].

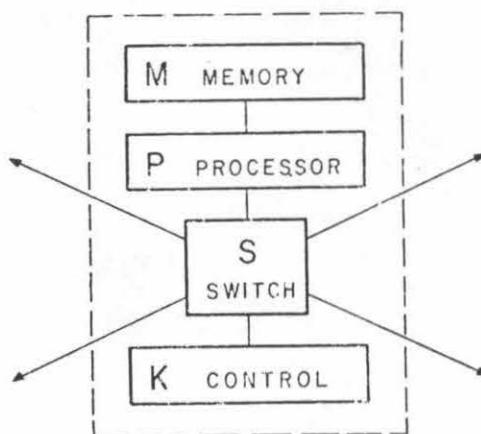


Figure 1. X-NODE, a modular VLSI building block of the mid 1980's, shown in PMS notation.

4. PROJECT X-TREE

In 1977 a research project to investigate these questions was started at the University of California at Berkeley. One specific issue was to define a modular component from which general purpose computing systems of arbitrary size could be built. Since the most advantageous way to interconnect these components turned out to be a tightly coupled, hierarchical, tree-structured network, the project was named X-TREE [Despain & Patterson 1978a]. And the projected single component, from which this computing system would be assembled, is known as X-NODE. The project is still in its early stages of research. First some of the communications issues have been addressed in a top-down manner [Séquin *et al.* 1978]. Discussions of the design of the architecture of X-NODE have recently been started [Patterson *et al.* 1979]. The anticipated timeframe for a potential realization of X-NODE is the mid-1980's. It is assumed that at that time VLSI chips with about 100'000 gates and half a million bits of storage will readily be feasible. Key research issues are the design of a truly modular VLSI component and the organization of many such components into a general purpose computing system.

5. INTERCONNECTION TOPOLOGIES

In an evaluation of different multiprocessor interconnection topologies [Despain & Patterson 1978b], it turned out that most schemes have one of three disadvantages.

- 1) The interconnecting link constitutes a serious communication bottleneck, such as a common bus which has to be shared by all attached processors.
- 2) The switching hardware becomes unreasonably expensive for a large number of processors, as in fully interconnected networks or in a full crossbar arrangement.
- 3) The interconnection scheme is not truly modular, because the requirements for the individual components change as the system grows. This is the case in a hypercube network with the topology of a cube in n dimensions, where the number of ports has to increase with the logarithm of the number of nodes.

Networks that are truly modular and incrementally expandable comprise lattice structures in n dimensions and tree structures. Among those, trees have the further advantage that the average distance between nodes grows only logarithmically with the number of nodes [Despain & Patterson 1978a,b].

The primary contender for the interconnection scheme in X-TREE is therefore a binary tree enhanced with additional links to form a half-ring or full-ring tree (Fig.2). These additional links further shorten the average path length, they distribute message traffic more evenly throughout the tree, and they provide the potential for fault tolerant communication with respect to the removal of a few nodes or links. Various placements for these extra links have been investigated, but most yield only insignificant improvement on the message traffic density and typically require significantly more complicated routing algorithms than the half-ring or full-ring structures [Despain & Patterson 1978b].

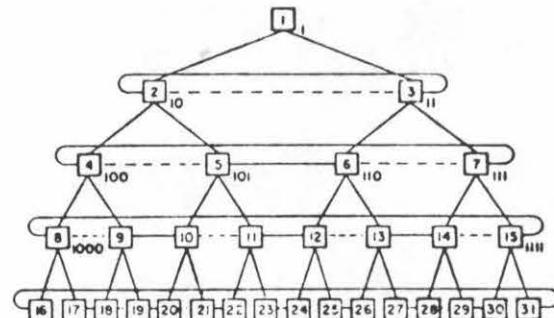


Figure 2. Binary tree with full-ring connections. When the dashed branches are omitted, a half-ring tree is obtained. Notice that the children of node n have node addresses $2n$ and $2n+1$ respectively.

6. ADDRESSING AND ROUTING

X-TREE is designed to be a truly modular, incrementally expandable system. To allow the system to grow to arbitrary size, neither the number of nodes nor the address space should be limited by artificial restrictions. Thus unbounded, variable length addresses are employed throughout [Séquin et al. 1978].

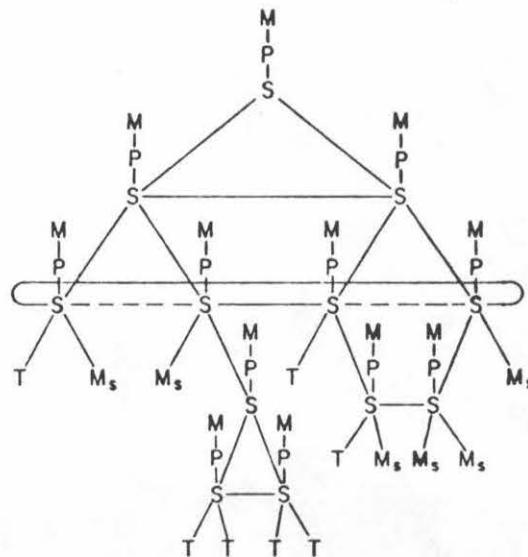


Figure 3. An X-TREE example in PMS notation.

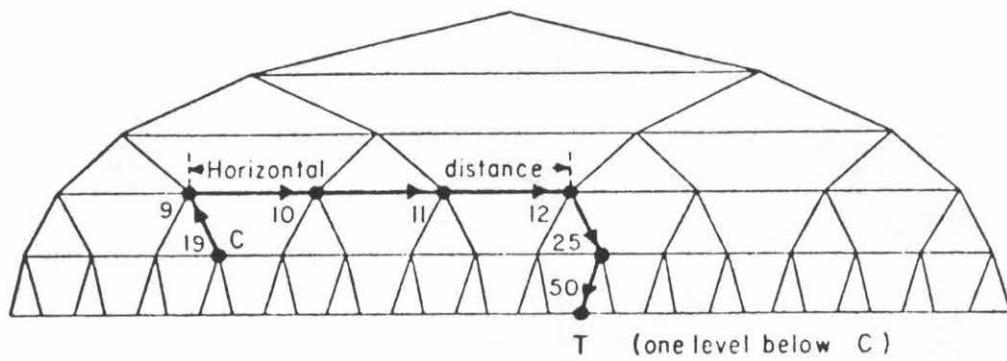


Figure 4. Shortest path in a full-ring tree from current position, C, on node 19 to target, T, on node 50.

All communication throughout the tree is in the form of messages. To enable effective routing of messages, the complete address is subdivided into a node address part and a second part identifying a particular memory location if that node has any memory attached that belongs to the global address space. In X-TREE secondary memory, as well as all input/output, is restricted to the frontier (i.e. the leaves) of the tree. This is to minimize the number of ports per node, and thus to alleviate the restrictions originating from the limited number of pins. A PMS representation of an example of a small X-TREE is shown in Fig.3.

The addressing scheme is designed so that messages can be effectively routed from node to node simply based on local decisions, involving only the current node address and the destination address. In a binary tree this can be achieved in a very simple manner. The root node is assigned node address "1". The node address of a left child is formed by appending a "0", and that of a right child by appending a "1". With that scheme any node can readily be found by starting at the root and using the sequence of bits in the node address to make the routing decision at each node. To go from an arbitrary node to another node, one has to move up in the tree to the common ancestor of the two nodes, i.e. to the node where the address matches all leading bits in the target address. From there one moves down to the destination.

In half-ring and full-ring binary trees the routing algorithm is only marginally more complicated. The horizontal links permit a message to take a shortcut before the common ancestor has been reached. It turns out that in a full-ring tree the optimal level to take the horizontal links is the one where ascending and descending path are separated by less than five link units (Fig.4). Again the routing decision can be based entirely on a comparison of the local node address and the target address.

Choice of directions as a function of relative target position							
choice		far left	near left	left line	right line	near right	far right
above	1 2 3	up left right	up left right	up left right	up right left	up right left	up right left
level	1 2 3	up left right	left up ldown	proc. back back	proc. back back	right up rdown	up right left
below	1 2 3	up left ldown	left up ldown	ldown rdown left	rdown ldown right	right up rdown	up right rdown

Table 1. Decision table for full-ring tree routing algorithm

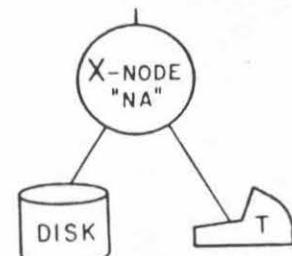
Some fault tolerance can be achieved by listing secondary and tertiary choices for the routing decision, which are taken when

the primary choice cannot be followed. Such a decision table for the case of the full-ring tree is shown in table 1. The proper field is selected from a comparison of the horizontal and vertical distances between current position and target node. Within each field, first, second and third choice for the routing out of the current position are listed.

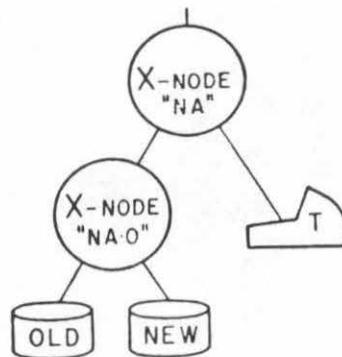
Of course there are always cases where this fault tolerant routing algorithm cannot be successful - for instance when the target node itself is missing. Special means to safeguard against cluttering the tree with worthless messages are thus required. One possible solution is to accompany each message header with a byte that counts the number of detours experienced, i.e. how often the message could not be routed in the primary direction. When this count reaches a limit, preset by the operating system, a higher level recovery mechanism is invoked. For instance, the message could then be purged and a notice of this fact could be returned to the originator.

7. SYSTEM EXPANSIBILITY

In X-TREE all input or output and secondary memory is at the frontiers of the tree. Thus, every time a node is added to the tree, a terminal or a storage device has to move and thereby changes its node address. A mechanism is necessary to route messages automatically to that node even though the message header may still carry an outdated node address. With the following conventions this can be achieved. Messages for leaf nodes are identified as such. Each node knows whether it is a leaf node or not. Messages destined for leaf nodes are routed downwards from their specified target node address along the chain of left children until they reach an actual leaf node. This scheme works, if during the expansion of the tree existing leaf attachments are moved to the left child position, of the newly attached node, and the right child position is used to attach new equipment (Fig.5).



before expansion



after expansion

Figure 5. Principle of incremental expansion of X-TREE, which permits messages to find the proper leaf node.

8. MESSAGES

If X-TREE lives up to its potential of substantial parallel computing, many messages will travel simultaneously through the tree, and many messages may want to use the same link between two nodes. In order not to stall one message until another message has been transmitted completely, messages are time multiplexed on the links on a demand bases. For this purpose each message header carries a unique identifier called a "slot address", so that the various parts belonging to different messages can be sorted out again at the receiving node. In each node all incoming messages are assigned such a "slot address", which is valid within that node and on the subsequent link. A specific slot address precedes any part of a message transmitted over a particular link.

Bytes	Explanation
SA / new	Slot address (SA) of type "new"
Target NA	together with target node address (NA), which may extend over several bytes, sets up a new message channel through the tree.
NA cont.	
Data	Beginning of first submessage.
Data	
:	
SA / old	Here, one or several other messages may be interspersed on the same link.
Data	Slot address of type "old" identifies continuation of a previous message.
Data	Data may consist of one or more submessages which may be individually checked and acknowledged to the originator.
Data	
:	
SA / old	Here, one or several other messages may be interspersed on the same link.
Data	Previous message continues.
Data	End of last submessage in this channel, may include crc check remainder and end of transmission character.
SA / end	Tear-down slot address removes message channel.

Table 2. Message format.

For the purpose of creating and removing message paths, there are different types of slot addresses. With the header of a new message, which carries the target address, a slot address of type "new" is transmitted, to establish a path throughout the tree to the proper destination. All subsequent bytes are considered to be part of the message content and are thus not involved in the routing process. If a message has to be interrupted, then subsequent parts of that message will be identified with the same slot address as before, but of type "old" to indicate that this is a continuation of a previous message. Active transmission is terminated by the originator by sending a slot address of type "end", which removes the established message channel. These conventions result in the message format shown in Table 2.

9. X-NODE ARCHITECTURE

At a first glance, each X-NODE is simply a computer that communicates with 4 or 5 nearest neighbors. However because of the bandwidth requirements discussed above, normal microprocessor input/output techniques are inadequate. It is important that the processor is not involved in the menial task of routing messages through X-TREE. Computation must occur in parallel with communication. Thus each X-NODE contains a self controlled switching network with its own I/O buffers and controllers. In the simplest version of X-NODE, the actual processor is attached to this network in the same manner as the links to the nearest neighbors (Fig.6). This permits an easy separation of the development of the communications hardware and of an advanced X-NODE computer.

10. SWITCHING HARDWARE

The heart of this switching network is a time multiplexed bus. Since this bus is anticipated to be completely contained within one chip, its associated parasitic capacitances will be rather low, and the resulting bandwidth for a given amount of drive power will thus be about an order of magnitude higher than that through the package pins associated with the input / output ports. This bus can thus conveniently serve the attached six to eight ports at their full capacity (Fig.6).

Each port consists of a set of input and output buffers and the necessary finite state machines to control them. Arriving slot addresses, which precede every separate part of a message and are identified by a special tag, switch the input multiplexer to the proper fifo buffer for that particular message (Fig.7). The internal communications bus consists of a data bus and an address bus carrying port numbers as well as slot addresses. The combined bus is allocated in a fixed round-robin manner to all attached input ports, which in turn can transmit one byte to a particular output port or to the routing controller.

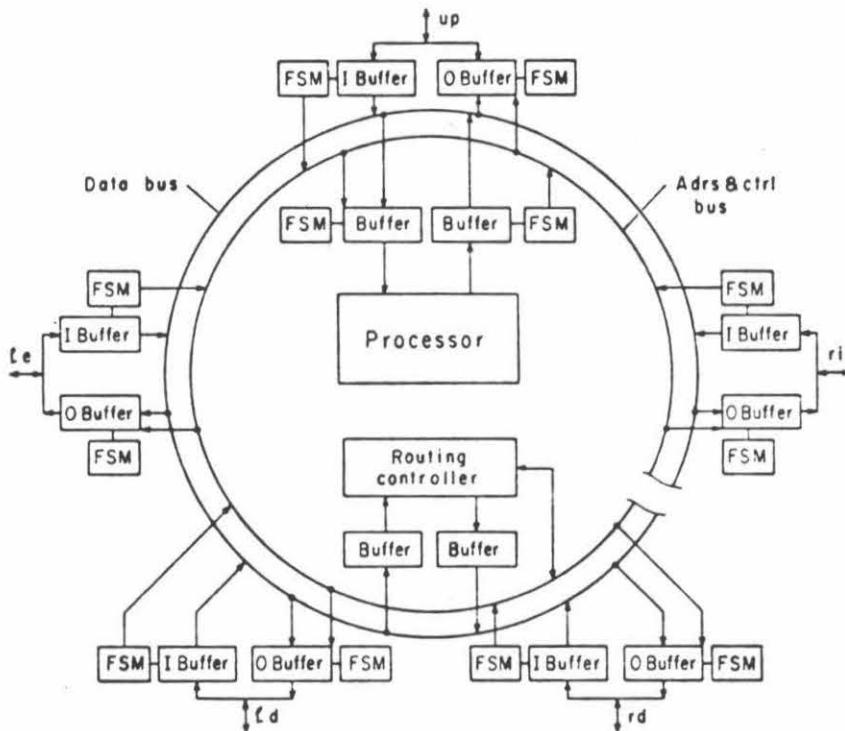


Figure 6. Block diagram of X-NODE, showing the X-NODE processor and the switching network consisting of intra-node bus, routing controller and input/output buffers to the communication links.

Slot addresses of type "new" cause the message header to be sent to the routing controller. There the proper output port is determined from the routing algorithm and a suitable slot address will be assigned to this newly to be established message channel. This information is returned to the port of entry where it is written into a look-up table (Fig.7). From then on, the intra-node communication for this channel can be handled directly by the input port.

Each output port monitors the intra-node bus. If it finds its own port number on the address bus, it will pick up a slot address and the corresponding data byte and enter the later in the proper fifo output buffer (Fig.8). Simultaneously, the finite state machine at the output end selects a channel with valid data for transmission over the link. The data bytes are preceded by the transmission of the corresponding slot address.

11. X-NODE MEMORY

Memory contained within each X-NODE is not part of the global address space. It acts only as a cache to the secondary memory contained entirely at the leaves of X-TREE. To alleviate the paging overhead, this memory should be as large as possible and thus the densest storage technique should be employed. For the mid 1980's it is anticipated that about 64k bytes could be implemented together with the X-NODE processor on the same chip if dynamic RAM or charge coupled devices are used. The contents of this local main memory can be data, program or microcode, and they will thus be used by different functional blocks. Unfortunately, typical high-density memory cannot easily be implemented as a true multiport memory, and thus it is not possible to extract three words from different locations simultaneously. Severe contentions for the memory bus can thus be expected.

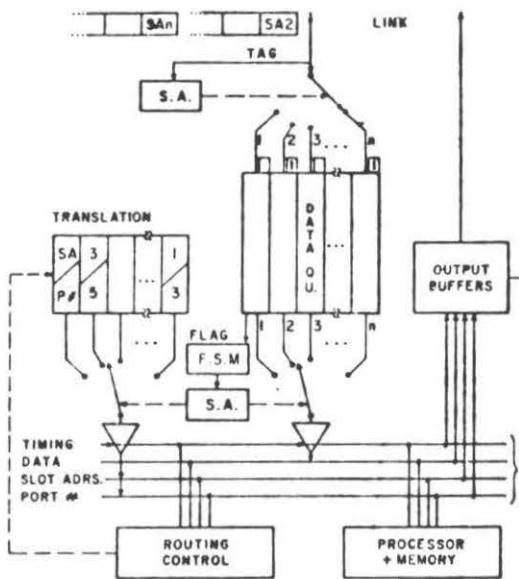


Figure 7. Blockdiagram of input port.

To alleviate this problem, we have decided to build an on-chip memory hierarchy consisting of a high-density RAM and three high-speed static caches dedicated to data, instructions and microcode, and therefore closely tied in with the ALU, the instruction decoder and the microcontroller, respectively. Contentions for the common data path between the main memory and the three caches can be minimized by proper choice of the cache parameters. This approach combines the storage density advantage of dynamic RAM with the higher speed of static caches and yields a lot of flexibility in the allocation of local main memory space to data, programs or microcode.

Swapping of microcode occurs through the same mechanisms as paging of programs or data from secondary memory at the frontier of X-TREE. No special mechanism has to be invoked to change the instruction set of one of the X-NODES, and the latter can thus be tailored dynamically to best solve the current computational problem.

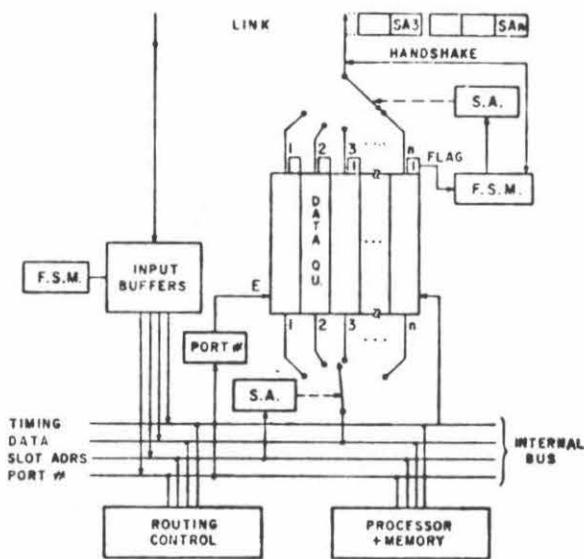


Figure 8. Blockdiagram of output port.

12. OTHER ARCHITECTURAL FEATURES

There are other architectural features which indirectly relate to the communication between processors. Issues of timing, synchronization, data sharing and protection are all crucial in multiprocessor, multi-user systems. The architecture of future chips should thus lend ample support to those features. Specifically, the concepts of processes [Dijkstra 68, Wirth 1971, Brinch Hansen 1972, 1975] and modules [Wirth 1977] are important for a clean and structured approach to parallel computing. In X-TREE we plan to give strong support to the mechanisms that create, terminate or move processes throughout the system [Patterson *et al.* 1979]. A vital part in this context is a mechanism that permits fast process switching without the explicit need to save many special registers. Again the cache based architecture provides a fast, automatic and transparent mechanism for the replacement of the active parts of the memory contents when a process environment is changed. A

small operating systems kernel based on the language Modula [Wirth 1977] is currently being developed and will be used in future extensive studies of the operating issues.

It goes almost without saying, that the customers of the next decade will expect strong support for high level language constructs, data structures, bounds checking and any thing that may help to slow down the increase of the gap between software and hardware costs.

13. SUMMARY

In summary, there will always be a demand for off-the-shelf, general-purpose components, which can be used as truly modular building blocks for the construction of computing systems of any size. One such component has been identified and one way to organize that component into an incrementally expandible computing system has been outlined. Two crucial issues in the design of such a system are interprocessor communication and the operating system. Both issues must be addressed at an early stage of the design so that the necessary hardware support can be provided.

We felt that the issue of interprocessor communication is even more important at an early stage of a VLSI multiprocessor project. The emergence of truly modular, general purpose VLSI components may actually depend on the design of a standard interprocessor communication protocoll and a realization of the required switching hardware on VLSI chips.

14. ACKNOWLEDGMENTS

The author would like to point out that project X-TREE is a "tightly coupled" team effort of our Architecture Group in the Computer Science Division at Berkeley, including Profs. Despain and Patterson, and several graduate students, and that it is almost impossible to determine the specific contributions of each member of the team. Particular thanks go to Al Despain and Dave Patterson for their suggestions, comments and careful review of this manuscript.

This study was sponsored in part by the Joint Services Electronics Program, Contract F44620-76-C-0100.

REFERENCES

- Bell,C.G. and Newell,A. (1971):
in *Computer Structures: Readings and Examples*, McGraw-Hill 1971, Chapter 2.
- Bell,C.G., Chen,R.C., Fuller,S.H., Grason,J., Rege,S. and Siewiorek,D.P. (1973):
"The Architecture and Applications of Computer Modules: A Set of Components for
Digital Design", *IEEE Compcon*, March 1973, Conf. Proc. pp 177-180.
- Brinch Hansen,P. (1972):
"Structured Multiprogramming", *Comm. ACM* 15, No 7, July 1972, pp 574-578.
- Brinch Hansen,P. (1975):
"The programming language Concurrent Pascal", *IEEE Trans. Software Eng.* 1, No 2,
1975, pp 199-207.
- Despain,A.M. and Patterson,D.A. (1978a):
"X-Tree: A Tree Structured Multiprocessor Computer Architecture", *5th Symp. on
Comp. Arch.*, Palo Alto, CA, April 3-5, 1978, Conf. Proc. pp 144-151.
- Despain,A.M. and Patterson,D.A. (1978b):
"The Computer as a Component: Powerful Computer Systems from Monolithic
Microprocessors", submitted to *Comm. of ACM*.
- Dijkstra,E.W. (1968):
"Co-operating sequential processes", in *Programming Languages*, ed. F. Genuys,
Academic Press, London, 1968.
- Haendler,W., Hofman,U. and Schneider,H.J. (1976):
"A General Purpose Array with a Broad Spectrum of Applications", in *Computer architecture workshop of the Gesellschaft fuer Informatik Erlangen*, May 1975, Informatik
Fachberichte, Springer, Berlin, 1976.
- Patterson,D.A., Fehr,E.S. and Séquin,C.H. (1979):
"Design Considerations for the VLSI Processor of X-TREE", submitted to the 6th
Annual Symposium on Computer Architecture, Philadelphia, April 1979.
- Séquin,C.H., Despain,A.M. and Patterson,D.A. (1978):
"Communication in X-TREE, a Modular Multiprocessor System", *ACM 78*, Washington
D.C., Dec.4, 1978, Proc. pp 194-203.
- Wirth,N. (1971):
"The programming language Pascal", *Acta Informatica* 1, 1971, pp 35-63.
- Wirth,N. (1977):
"Modula: A language for modular multiprogramming", *Software - Practice and Experience* 7, No 1, 1977, pp 3-35.

A CELLULAR, LANGUAGE DIRECTED COMPUTER ARCHITECTURE
(Extended Abstract)

Gyula A. Magó

University of North Carolina at Chapel Hill

Abstract If a VLSI computer architecture is to influence the field of computing in some major way, it must have attractive properties in all important aspects affecting the design, production, and the use of the resulting computers. A computer architecture that is believed to have such properties is briefly discussed.

I. Introduction

One would expect that microelectronics, having affected other application areas, should influence the way we design computers, and such sentiments have already been articulated in the literature [5]. So far, however, all ideas on how to organize a very large number of logic circuits (or for that matter, microprocessors) into a computing machine have had serious drawbacks in one or more respects. As a consequence, computer design has not yet been able to exploit successfully the ever-increasing capabilities of semiconductor technology.

In the first part of this paper, we argue that a VLSI computer architecture, in order to have a major impact on the computing field, (1) should be cellular in design, and (2) should have its design be guided by consideration of an appropriate language. The remainder of the paper outlines the major characteristics of one computer architecture which has these properties. A detailed description of this architecture is being published elsewhere.

II. Desiderata for VLSI Architectures

We start this sequence of arguments by acknowledging that for the foreseeable future, only large production volumes can make it economically attractive to manufacture whatever VLSI chips we need for our computer designs. From this point of view, all conceivable computer designs requiring Q amount of hardware for their realization (measured in any meaningful way) can be linearly ordered: at one end of the scale one finds designs that require the whole Q amount of hardware to be designed into different, one-of-a-kind chips,

whereas moving towards the other end one finds designs (we shall call them cellular) in which only a fraction, Q/n amount, of the hardware has to be designed into chips, and the computer is obtained by taking n copies of these chips and interconnecting them in some regular fashion. If one can create sensible computer designs that are cellular, then, all other things being equal, increasing n will make the situation more and more attractive by decreasing the number of necessary chip designs and increasing production volume of each remaining chip type.

We infer from the above that a VLSI computer architecture should have a cellular structure, that is, it should be obtained by interconnecting simple component processors (we shall call them cells) in a regular fashion. (At this stage we can conceive of a cell occupying more than one chip, or alternatively a chip containing many cells. In this latter case, the chips should also be connected in a regular fashion.) We shall take cellularity also to imply that in an organizational sense such an architecture should be expandable, and that there should be only physical limits to the size of such machines.

In a cellular computer, the cells are expected to operate concurrently, carrying out component computations, and the problem of designing such a computer is ultimately a problem of algorithm decomposition: how to prescribe for each cell what to do so that the totality of their behaviors adds up to the required global behavior, such as the execution of a particular user program. Since computations in a cellular computer must be able to unfold over potentially very large collections of cells (thousands or millions of them), often in a data dependent manner, it would be most natural for the details of the above mentioned decomposition to be worked out at run-time. The remaining question is how is the decomposition determined and at what cost?

The inherent nature of a cellular computer--it is expandible, and arbitrarily large collections of cells must be able to operate efficiently--appears to exclude any reliance on some central, global agent, e.g., a "master" computer, that would inspect the user program, decompose it, and then determine what each cell should do. The alternative to such a global control is to let individual cells cooperate in decomposing the user program into parts (small or large), carry out partial computations, and (still using only limited local information) cooperate in combining the partial results to obtain the desired overall result. The more straightforward, direct, and simple this process can be made, the more efficient the resulting computer can become. In the extreme case, the cellular network can be thought of as directly executing an appropriate user language. It appears most promising then to require that a VLSI computer architecture be language directed, meaning that it be able to execute directly a programming language. With this approach, the programming language specifies the global behavior the cellular network is to exhibit, which in turn can be used to derive the behavioral and structural specifications for the individual cells. The above requirement really urges an integrated, top-down design for both the software

and the hardware of such a cellular computer, in sharp contrast with the usual practice of separately designing software and hardware for uniprocessors. (It should be mentioned that Dennis [3] and other advocates of the data flow approach have also argued for language directed architectures in the context of parallel, though not necessarily cellular, computers.)

In addition to the above considerations, an acceptable computer design for VLSI technology must meet many other criteria, such as ease of programming, efficient execution of user programs, and cost-effectiveness in an overall sense. In [4], a computer architecture is described that seems to meet these criteria. In this paper, we explore what a cellular, language directed architecture can offer by examining how some of these criteria are met by the architecture described in [4].

III. Main Characteristics and a Brief Evaluation of a Cellular, Language Directed Architecture

The architecture described in [4] is capable of directly executing certain applicative languages (e.g., reduction languages [1], FP and FFP systems [2]) recently introduced by Backus. In these languages, programs are expressions, and a program is executed by evaluating its expression. A particular class of expressions, called applications, specifies computations. Because these languages allow all innermost applications to be evaluated simultaneously, they are able to express parallelism in a very natural fashion.

The architecture of [4] is obtained by interconnecting cells in the form of a full binary tree, and additionally connecting the leaf cells into a linear array. The leaf cells (i.e., those in the linear array) are all identical and are called L cells. The remaining cells are different from the L cells but identical to each other; these are called T cells. The L cells store the expression to be evaluated (although they also have some processing capabilities), whereas the T cells are used for routing and processing purposes. In order to make the cells as small as possible, each L cell is used to store a single symbol of the source program.

A user program and its data form a single expression in these languages; this expression is a linear string of symbols, and is mapped onto the L array, one symbol per L cell. The innermost (hence executable) applications are contained in disjoint segments of the L array, and disjoint portions of the T network are used to evaluate (i.e., execute) them. Consequently, this cellular architecture is capable of unbounded parallelism on the source language level by simultaneously evaluating all innermost applications, the only limitation being the size of the L array.

In addition, there is parallelism below the level of the source language: since cells of L hold only single symbols of the source language, even the simplest primitive operations of the source language, such as adding two numbers, involve the cooperation of several cells of the network. This low-level parallelism makes feasible direct implementation of complex primitive operations of the language, such as vector operations and typical associative processor operations.

Since the above architecture is unusual, one can evaluate it or compare it with other architectures only on the basis of global, overall criteria, for ultimately its viability will be judged on the basis of such criteria. We consider four main categories of properties of the architecture.

1. Programmability

1.1. The applicative languages on which the architecture is based support a functional style of programming. The advantages of such a style over that permitted by conventional languages are discussed at length by Backus in his 1977 Turing Award Lecture [2].

1.2 These languages allow and encourage the introduction of a high degree of parallelism into the programs without requiring detailed planning on the part of the programmer (for example, the programmer does not have to initiate and terminate execution paths explicitly).

1.3 Execution times of programs can often be predicted analytically (see 2.). As a result, when a program is being written, efficient execution can be a criterion, and time-space tradeoffs are commonly available.

1.4. The primitive operations of the language are not wired into the cells, and consequently the architecture allows great flexibility as far as the primitives of the applicative languages are concerned, even within a single machine.

2. Efficiency of Program Execution

The processor is designed to be able to execute with acceptable efficiency any program written in an applicative language. Most important in this respect is that the processor initiates and terminates the execution of all innermost applications (i.e., execution paths) with little or no overhead (beyond the work required by individual innermost applications), thereby adjusting easily to the widely varying degrees of parallelism found in most programs.

The execution time of an innermost application can be predicted analytically. Some primitive operations require $O(\log N)$ time, where N is the number of cells in L. More complex primitives, such as reversing a list of an arbitrary number of elements, require time proportional to the number of data items that have to be moved. Based on the execution times for primitives, upper and lower bounds can be derived for the execution times of programs written in the applicative language; this is especially easy for programs with few

data dependent branches. Many important classes of algorithms have been and are being analyzed, such as vector and matrix algorithms, algorithms for dense and sparse linear systems, Fourier transforms, solution methods for partial differential equations, and multidimensional search problems. These analyses show that not only can the machine execute sequential programs in an acceptable manner, but also the massive parallelism both on and below the source language level can result in greatly increased execution time efficiencies. The details of these analyses are to be published elsewhere.

3. System Software

Since the cellular network directly responds to an applicative language, the need for many typical components of present-day uni-processor and multiprocessor software has been removed, and their function taken over by hardware. For example, in the presently envisioned system, there is no need for

- (a) a compiler (there is only need for a very simple preprocessor to change the external representation of programs to an internal one),
- (b) a software interpreter,
- (c) memory management software,
- (d) software to detect parallelism in user programs,
- (e) software to assign processors to tasks.

Since the cost of software is the dominant component in the cost of present-day general purpose computers (and not only is it expensive to develop such software, but in addition it has to be stored in the machine, and it ties down hardware resources while executing), the possibility of trading complex system software for cellular hardware is a very welcome development.

4. Hardware Related Issues

Some of the advantages of cellularity have already been discussed under the desiderata.

4.1. Both the L and T cells are rather small, mainly because they need only a few dozen registers as local storage. As a result, as VLSI technology advances, whole subtrees of cells may be put on a single chip. (Any cell of T or L has to be connected to at most three other cells in the network. Furthermore, any complete subtree whose leaves are L cells communicates with the rest of the processor through at most three such points.)

4.2 Since a collection of user programs is again an expression in the applicative language, the cellular processor needs no extra machinery to deal with several user programs at a time. As a result, increasing the size of the processor can yield arbitrarily large throughput values without any need for reprogramming. (In fact, all other things being equal, throughput increases at least as fast as $N/\log N$, where N is the number of L cells.)

4.3 Actual throughput values are influenced by many design parameters, such as speed of logic circuits and widths of data paths but most importantly by the parallelism present in user programs.

For programs that offer high degrees of parallelism, throughput values can become very high. As an example, for $N=1$ million L cells, 5,000 - 10,000 MIPS can realistically be estimated as the highest attainable, where an innermost application is counted as an "instruction." (Note that if an innermost application involves a complex primitive, such as finding the maximum of an arbitrary number of data items, what we count as one instruction may be equivalent to the computational work performed by many uniprocessor operations.)

References

- [1] J. Backus, Programming Language Semantics and Closed Applicative Languages. IBM Research Report RJ1245, Yorktown Heights, N.Y. July 1973.
- [2] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 613-641.
- [3] J. B. Dennis, "Programming Generality, Parallelism and Computer Architecture," Information Processing 68, North-Holland Publishing Co., 1969, pp. 484-492.
- [4] G. A. Magó, "A Network of Microprocessors to Execute Reduction Languages," International Journal of Computer and Information Sciences (to appear)
- [5] I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," Scientific American, Vol. 237, No. 3, September 1977, pp. 210-228.

Computations on a Tree of Processors

Sally A. Browning
Computer Science Department
California Institute of Technology
Pasadena, California 91125

Because processors and memories are both implemented in silicon, it is worthwhile to consider architectures that mingle both functions on a single chip. With the VLSI promise of a million or so devices on a chip, several hundred processors can communicate with each other at on-chip speeds.

But in order to manage the complexity of such a chip, both when designing it and when testing it, the interprocessor communication paths should be regular and simple. This paper examines the utility of a particular interconnect scheme, a binary tree.

The processors are arranged as a binary tree: each processor except the root has a single parent, and each processor except those at the leaves of the tree has two descendants. This arrangement models the hierarchical communication found in large organizations.

The binary tree architecture has some interesting aspects that make it a good choice for a general purpose structure. Any particular processor in a tree of n processors can be accessed in at most $\log_2 n$ time. This compares favorably with the $O(n)$ access time for a linear arrangement of processors, or the $O(\sqrt{n})$ access time if the processors are arranged in a rectangular array.

The number of processors available increases exponentially at each level in the tree machine. If the problem to be solved has this growth pattern, then the tree geometry will match the problem. By

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency under contract number N00123-78-C-0806.

contrast, processors arranged as a list have a constant number of processors (namely 1) available at each level. And rectangular arrays make a polynomial number of processors available at each level, according to the allocation scheme chosen. Figure 1 demonstrates this property of the three structures.

These three schemes are the simplest ways to connect processors together. They provide each processor with two (the list), three (the tree), or four (the square array) neighbors. Each has advantages over the others, and each has a fan club.

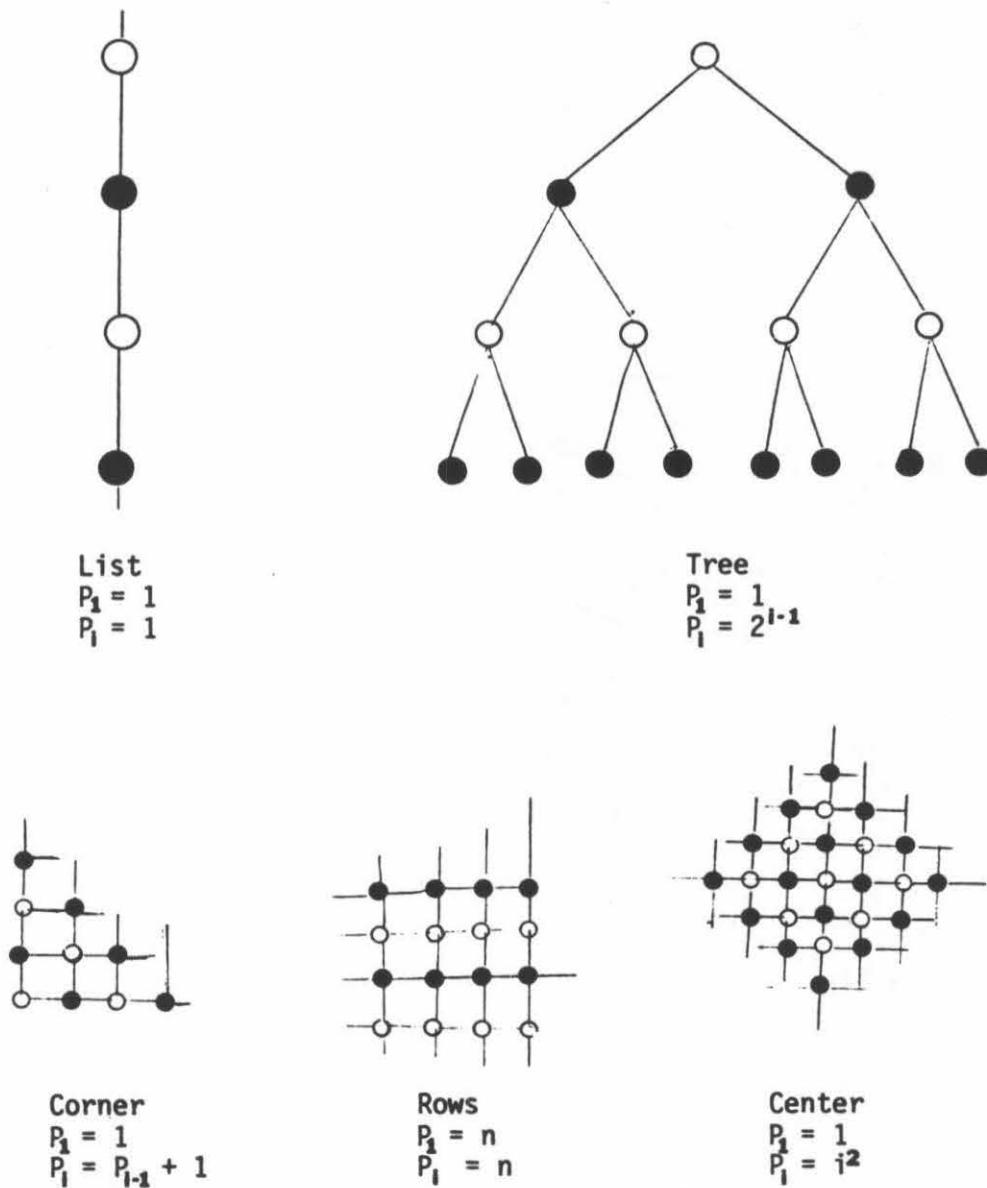
The point of my research is to determine whether or not there is a predominate geometry to the problems that might be solved on a highly concurrent machine. If such a geometry exists, and a hardware implementation is realizable in silicon, then that machine should be built.

Since the tree architecture appears to have more flexibility than the other two structures, I have concerned myself mostly with it. This paper will describe several algorithms that have been successfully mapped onto the tree. In later sections, the Ringmachine, a linear array of processors proposed by Mike Ullner[7], will be introduced in order to show that problems that are dominated by loading and unloading do not require the additional communication paths available in the tree. The final section of the paper describes a problem from numerical analysis that makes effective use of the tree machine. The paper ends with some comments about the direction future investigations will take.

A Digression into Programming Notation.

The processors in the tree have some characteristics that must be emphasized by the notation used to describe them.

First, each processor is a general computing machine with some amount of local store. A template that describes both the program



P_i = number of processors at level i .

Figure 1. Processors Available at each Level.

and the data that will characterize each processor. This template will be instantiated as many nodes of the tree.

Communication between each processor, its parent, and its children should be limited to explicitly defined entry points. That is, there is no omnipotent entity that is able to oversee and influence the actions of other processors except as explicitly described. Each processor can expect to have local sovereignty, and can only be affected by communication it expects.

And perhaps most importantly, locality should be encouraged in the problem solutions. Communication between processors requires synchronizing their actions, limiting the amount of concurrency that can be achieved.

The notation that embodies these criteria is the class construct described by Dahl and Hoare [4]. The class allows the programmer to define as a single entity both a data structure and the procedures that operate on it. Thus the implementation details are known only to the class itself. Each object is an instance of a class, and can be thought of as a machine, capable of local computation but responding to well-defined requests from the outside world.

The most widely known programming language that incorporates the class construct is SIMULA 67 [2]. SIMULA extends the syntax of Algol 60 with class definitions. I will use a modified version of the SIMULA syntax to describe the nodes in the processor tree. The syntax for a class declaration can be described in BNF as follows:

```
<class declaration> ::= class <class identifier>
                           <formal parameter part>;
                           <attribute part>;
                           <class body>

<class body> ::= <statement>
```

In order to describe highly concurrent algorithms despite the sequential nature of SIMULA, the meaning of the semicolon symbol is changed. In vanilla SIMULA, semicolon is used to terminate a statement. Instead, read semicolon as "At this point, all statements in progress must be terminated before advancing to the next statement". Linefeed will be used to indicate syntactic end of the statement. In other words, linefeeds are used to separate statements; semicolons are used to separate groups of statements that can execute concurrently. E. W. Dijkstra introduces this semicolon convention in [6].

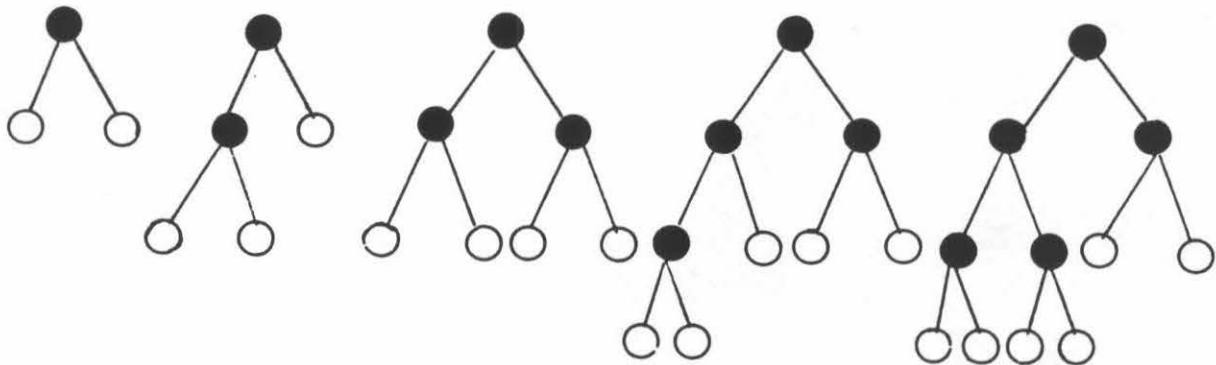
Making Arbitrary Branching Ratios.

While the physical structure of the tree restricts each processor to two descendants, a logical structure can be imposed on the tree to accomodate an arbitrary branching ratio. Each logical processor consists of several physical processors, enough to provide the desired number of offspring. A logical node with n children is built from $n-1$ physical nodes and is $\log_2 n$ levels deep. Figure 2 shows some sample logical processors. Figure 3 gives a SIMULA representation of the algorithm used to simulate arbitrary branching ratios.

All of the algorithms described in this paper will describe logical processors and the logical structure of the tree. The SIMULA code assumes the existence of the logical processor defined in Figure 3, and builds definitions based on it. The complexity of each algorithm will be calculated for the physical structure, however.

Sorting.

A binary tree with depth $\log_2 n$ can be used to sort n numbers. The sort is accomplished as a byproduct of loading the numbers into memory and then reading them out again. The numbers themselves are



**Figure 2. Logical Processors with 2 to 6 Descendents
(solid color boxes comprise the logical processor)**

```

CLASS Node(n); INTEGER n;
BEGIN
  REF(Node)left,right;

  !init code to build logical node;
  IF n>2 THEN left:=NEW Node((n+1)//2);
  IF n>3 THEN right:=NEW Node(n//2);

END of CLASS Node;

Node CLASS Processor;
BEGIN

  REF(Processor) PROCEDURE Son(s); INTEGER s;
  BEGIN REF(node)p;
  p:=IF s<=(n+1)//2 THEN left ELSE right;
  WHILE NOT (p IN Processor) DO
    p:=IF s<=(p.n+1)//2 THEN p.left ELSE p.right;
    Son:=p;
  END of PROCEDURE Son;

END of CLASS Processor;

```

Figure 3. Making Arbitrary Branching Ratios

never in sorted order internally, but come out of the tree in the desired order.

Sorting is a particularly interesting example because it illustrates a fundamental issue in concurrency. It is well known that sorting on a sequential machine can be done with $O(n \log_2 n)$ comparisons. However, it has been shown on very fundamental grounds that if communication is restricted to nearest neighbors, at least n^2 comparisons are required[5]. The apparent advantage of the $O(n \log_2 n)$ algorithms comes as a direct result of longer communication paths. It is also clear that no scheme will be able to produce an ordered set of numbers until all numbers are loaded into the machine. This means that the best achievable time complexity is $O(n)$.

The algorithm I use is an implementation of heap sorting. The algorithm that runs in each processor, given in Figure 4, has a procedure for loading the tree called Fillup and a procedure invoked during the output cycle called Passup.

Fillup keeps the largest number seen to date, and passes the smaller one to the right or left child, keeping the tree balanced by alternating sides.

Passup returns this processor's current number and refills itself with the larger of the numbers stored in its descendants. This action is pipelined so that the largest number is always available in the root.

This sort algorithm is bounded by the time it takes to load and remove the numbers. Thus it has time complexity $O(n)$. It requires n processors, one for each number to be sorted.

```

Processor CLASS HeapSort;
BEGIN

    INTEGER number;
    BOOLEAN balanced,empty;
    REF(processor)left,right;

    PROCEDURE fillup(candidate); INTEGER candidate;
    BEGIN
        IF empty THEN
        BEGIN
            number:=candidate
            empty:=FALSE;
        END
        ELSE
        BEGIN
            IF candidate>number THEN !swap;
            BEGIN INTEGER t;
                t:=candidate;
                candidate:=number;
                number:=t;
            END;
            IF balanced
            THEN left.fillup(candidate)
            ELSE right.fillup(candidate);
            balanced:=NOT balanced;
        END;
    END of procedure fillup;

    INTEGER PROCEDURE passup;
    BEGIN
        passup:=number;
        IF left==NONE AND right==NONE THEN empty:=TRUE !its a leaf;
        ELSE
        IF left.empty THEN
        BEGIN
            IF right.empty THEN empty:=TRUE !both subtrees empty;
            ELSE number:=right.passup; !fill from right son;
        END
        ELSE
        IF right.empty THEN number:=left.passup !fill from left son;
        ELSE number:=IF left.number>right.number
        THEN left.passup ELSE right.passup;
        !take the larger of the two;
    END of procedure passup;

    !init code;
    empty:=TRUE;
    balanced:=TRUE;
    !left and right set;

END of class HeapSort;

```

Figure 4. Heap Sort

Matrix Multiplication.

Consider the problem of multiplying two $n \times n$ matrices together. The tree machine algorithm that provides the answer in the least amount of time divides the multiplicand into rows and the multiplier into columns, pipelines the loading and multiplication of pairs of single elements. This process requires $O(n^2)$ processors and takes $O(n^2)$ time, a processor and time product of $O(n^4)$. If each processor has enough memory to store a row of the matrix instead of a single element, the algorithm would require $O(n)$ processors, resulting in the more familiar $O(n^3)$ product.

The algorithm makes use of a tree that has a branching ratio of n at each node, and is two levels deep. The root node has n descendants, each controlling n leaves of the tree. Then there are n^2 leaves and a total of $2n^2 - 1$ processors.

Each child node of the root, hereafter called a **row supervisor**, will represent a row of the multiplicand matrix, and produce a row of the product matrix. Each of the n descendants of a row supervisor will hold one element of the row.

The algorithm is given in Figure 5. The multiplier matrix is loaded into the tree one element at a time, by column. The root hands each element to all row supervisors, which send it to their appropriate leaf: the first element in any column goes to the first child of each row supervisor, the n th element to the n th child. That child multiplies the multiplier element by the multiplicand element it holds, and returns the product to the row supervisor. When an entire column of the multiplier has been loaded into the tree, each row supervisor takes the n products generated in its children, adds them, and returns one element in the corresponding column of the product matrix. That is, when the first column of the multiplier has been loaded into the tree, the first column of the product matrix is available, and so on.

This process can be pipelined to take $O(n^2)$ time. Thus the time it

```

Processor CLASS Rowsupervisor;
BEGIN
!the matrix size, N, is an attribute of CLASS Processor, and is available
to us;

REAL product;
INTEGER count;

PROCEDURE Load(element); REAL element;
BEGIN
  count:=count+1;
  son[count].load(element);
  IF count=N THEN count:=0;
END of procedure Load;

REAL PROCEDURE Multiply(element); REAL element;
BEGIN
  count:=count + 1;
  product:=product + son[count].multiply(element);
  IF count=N THEN
    BEGIN
      multiply:=product;
      count:=0;
      product:=0.0;
    END;
END of procedure Multiply;

Initialization;
count:=0;
product:=0.0;
END of class Rowsupervisor;

```

```

Processor CLASS Leaf;
BEGIN

REAL rowelement;

PROCEDURE Load(element); REAL element;
BEGIN
  rowelement:=element;
END of procedure Load;

PROCEDURE Multiply(element); REAL element;
BEGIN
  multiply:=rowelement * element;
END of procedure Multiply;

END of class Leaf;

```

Figure 5. Matrix Multiplication

takes to load the n^2 elements of the matrices dominates the time complexity of the problem. Remember, however, that matrix operations are meaningless except in the context of the driving problem. The entries in the matrix are not so much loaded as generated, and the generation time may be less than $O(n^2)$. Care must be taken, however, to generate the matrix entries in the arrangement used by the multiplication algorithm; moving elements around in the tree is costly.

The Color Cost Problem.

This NP-complete problem is an adaptation of the K-colorability problem. Given an undirected graph G of n nodes and a set of n colors, each with an associated cost, find a minimum cost coloring of the graph such that no nodes sharing an edge are the same color.

There are n^n possible colorings of the graph. Evaluating them sequentially produces a solution in time $O(n^n)$. I present a parallel algorithm of order n^2 .

Each level in the processor tree represents the consideration of another node. That is, level one shows possible colors for the first node, level two colors the second node based on the choices made for at level one, and so on. I will describe the generation of the potential colorings.

Each processor, described in Figure 6, has an edge list called `edge` and a list of costs indexed by color number called `colorcosts`. There is an array called `coloring` that reflects the color choices for preceding nodes, and a boolean array called `colors` that is used to generate the possible colorings for this node.

The algorithm, given in procedure `color`, begins by assuming that all colors yield valid colorings. The array `coloring` is used to eliminate those colors that have been used to color nodes that share an edge with this node. This reduced set of colors, all of

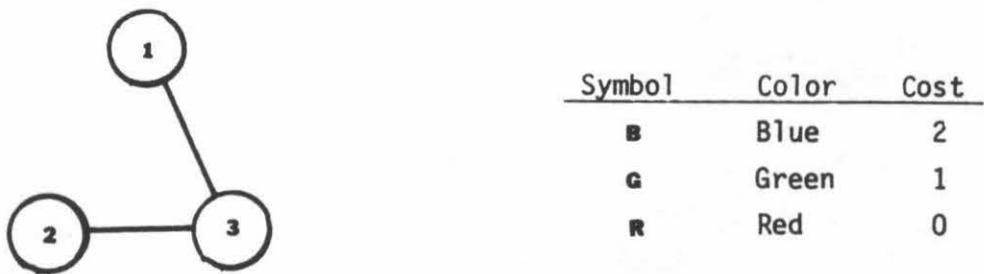
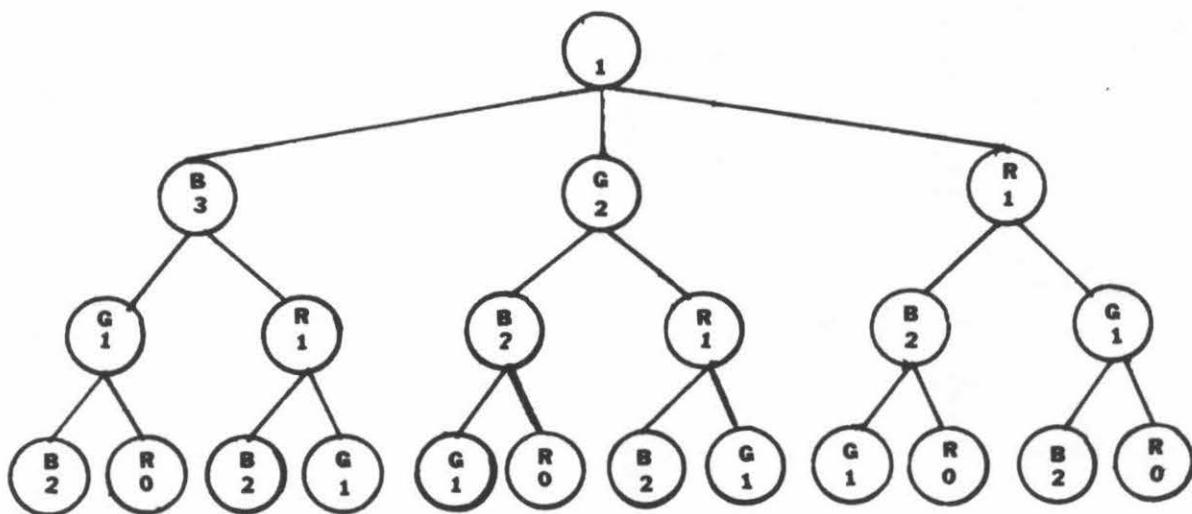
```
Processor CLASS ColorCost;
BEGIN

    BOOLEAN ARRAY edge[1:n,1:n],colors[1:n];
    INTEGER ARRAY coloring[1:n],colorcosts[1:n];
    INTEGER cost;

    PROCEDURE color(node); INTEGER node;
    BEGIN INTEGER i;
        IF node>n THEN
            BEGIN
                cost:=0;
                FOR i:=1 TO node-1 DO cost:=cost+colorcost[coloring[i]];
            END
        ELSE
            BEGIN
                FOR i:=1 TO node-1 DO IF edge[1,node] THEN
                    colors[coloring[i]]:=FALSE;
                FOR i:=1 TO n DO
                    IF colors[i] THEN
                        BEGIN
                            son(i).coloring[node]:=i;
                            son(i).color(node+1);
                        END
                    ELSE son(i):=NONE;
                    cost:=maxcost;
                FOR i:=1 TO n DO
                    IF (IF son(i) = NONE
                        THEN FALSE ELSE cost>son(i).cost)
                        THEN cost:=son(i).cost;
            END;
        END OF procedure color;

    END OF class ColorCost;
```

Figure 6. Color-Cost Problem

**Figure 7. Color-Cost Example: Graph and Color List****Figure 8. Color-Cost Example: Solution Tree**

which are legal colorings, is used to spawn descendants, one for each coloring of this node.

When the tree is n levels deep all the legal colorings have been generated. The leaf nodes calculate a cost for the coloring they represent, and each parent node takes as its cost the least cost among its children. Thus the minimum cost coloring is stored at the root.

An example is in order. A sample graph and color set are given in Figure 7. Figure 8 shows the colorings and costs arrived at by the algorithm. Each level of the tree represents a node of the tree. That is, if the root is level 0, the first node is colored in level 1, and level 3 represents potential colorings for the third node. Besides representing a part of a coloring, each node also contains the minimum cost coloring found among its descendent colorings.

The minimum cost of coloring the sample graph is 1, and is achieved by coloring nodes (1,2,3) (red,green,red).

When the color cost problem is solved in a brute force manner on a sequential machine, it takes exponential time. The tree machine can solve the problem in $O(n^2)$ time using an exponential number of processors. So on either machine, this problem exhibits exponential growth.

Transitive Closure.

Given a directed graph G , the transitive closure of G , G^* , can be generated. The arcs of G^* are subject to the following condition: for every arc (v,w) in G^* there is a path, $(v,e_1),(e_1,e_2), \dots (e_m,w)$, in G .

The best sequential algorithm for generating the transitive closure of a graph is attributed to Marshall[1,8]. The algorithm uses three FOR loops that run through the incidence matrix adding arcs. After

k steps of the outer loop, there is a path from vertex i to vertex j through vertices in the set $\{1, 2, \dots, k\}$ if and only if $B[i, j] = 1$. On a sequential machine, this algorithm takes $O(n^3)$ time. The code is given in Figure 9.

A direct mapping of Marshall's algorithm onto the tree machine yields a rather boring n^3 algorithm that merely spreads the three iterative steps among the processors in the tree.

There is a much more fruitful path to take. By understanding what actually happens during the execution of the algorithm, an effective mapping of Marshall's algorithm onto the tree machine is discovered.

There are two key points to be made about Marshall's algorithm. First, the algorithm is cascading. Newly created arcs can effect the creation of yet more arcs. Any realization of the algorithm must allow for this characteristic. It is not sufficient to consider only the arcs in the original graph.

Also important is the comparison between arcs. In Figure 9 this comparison is stated as

```
IF b[i,j] AND b[j,k] THEN b[i,k]:=TRUE;
```

In English, this reads "if there is an arc from i to j , and an arc from j to k , then create an arc from i to k ".

Suppose that instead of an incidence matrix, there is a list of arcs. This list will be used as input to the tree machine. The output is the list of arcs in the transitive closure.

The tree has a root node, n descendants of the root that are instances of the class vertex, and n^2 descendants of the vertex processors described by the class toVertex. The vertex processors represent the n nodes in the graph. The toVertex processors are the n possible arcs from each node. Jim Rowson deserves special thanks

for distilling my complicated structure into this very simple one.

The arcs in the original graph are used only as the starting place and are indistinguishable from generated arcs. As new arcs are created, they are considered by all the vertex processors just as the original arcs are.

Arcs are created using a variant of the Warshall comparison. An arc has a starting point, fromV, and an ending point, toV. Each arc is considered by all the vertex processors. Each vertex will create an arc by turning on its appropriate descendent if one of two conditions is true. Either this vertex must be the starting point of the arc, or there must be an existing arc from this vertex to the starting point.

The first condition takes care of the arcs in the original graph. The tree starts out with no arcs. As the original arcs are loaded into the tree, the first condition is true and arcs are created.

The second condition is the Warshall comparison. Suppose the arc (v,w) is being considered by vertex u. If arc (u,v) exists then arc (u,w) is created. This is how new arcs are created.

As each arc is created, by satisfying either criterion, it is broadcast throughout the tree; it might effect the creation of other arcs.

The code for this algorithm is given in Figures 10 and 11. Figure 10 shows the properties common to all three kinds of processor nodes, and defines some auxiliary classes used for queueing and passing data between processors. Figure 11 is the definition of the three processors, including the procedures that implement the revised Warshall algorithm.

The key routines are load and unload. Procedure load appears in the root and vertex processors and is used to pass arcs through the system. Unload is in the root. Each call on unload yields an arc in

```

BOOLEAN ARRAY B[1:n,1:n];
INTEGER i,j,k;

FOR k:=1 to n DO
  FOR i:=1 to n DO
    FOR j:=1 to n DO
      IF B[i,k] AND B[k,j] THEN B[i,j]:=TRUE;

```

Figure 9. Warshall's Algorithm (Sequential Machine)

```

CLASS processor;
BEGIN

  REF(processor) array son[1:n];
  REF(processor) parent;

END of class processor;

processor CLASS Qprocessor;
BEGIN

  REF(head)Q;
  PROCEDURE InsertInQ(qe); REF(queueElement)qe; qe.into(Q);

  REF(queueElement) PROCEDURE firstInQ;
  BEGIN REF(queueElement)qe;
    firstInQ:=qe:=Q.first;
    qe.out;
  END of procedure firstInQ;

  Q:=NEW head;

END of class Qprocessor;

link CLASS queueElement(myOwner); REF(Qprocessor)myOwner;
BEGIN
END of class queueElement;

CLASS edge(fromV,toV); INTEGER fromV,toV;
BEGIN
END of class edge;

```

Figure 10. General Processor Definition and Auxiliary Classes

```

Qprocessor CLASS root;
BEGIN
  PROCEDURE load(e); REF(edge)e;
  BEGIN INTEGER i;
    FOR i:=1 STEP 1 UNTIL n DO son[i].load(e);
  END of procedure load;

  REF(edge) PROCEDURE unload;
  BEGIN
    IF Q.empty THEN unload:=NONE
    ELSE BEGIN REF(queueElement)qe; REF(edge)e;
      qe:=firstInQ;
      unload:=e:=qe.myOwner.nextEdge;
      load(e);
    END;
  END of procedure unload;

  BEGIN Integer i;
    FOR i:=1 STEP 1 UNTIL n DO son[i]:=new vertex(i);
  END of init code;
END of class root;

Qprocessor CLASS vertex(myNode); INTEGER myNode;
BEGIN
  REF(queueElement)qe;
  BOOLEAN queued;

  REF(edge) PROCEDURE nextEdge;
  BEGIN REF(queueElement)qt;
    qt:=firstInQ;
    nextEdge:=NEW edge(myNode,qt.myOwner.myNode);
    IF NOT Q.empty THEN parent.insertInQ(qe)
    ELSE queued:=FALSE;
  END of procedure nextEdge;

  PROCEDURE load(e); REF(edge)e;
  BEGIN
    IF e.fromV=myNode OR son[e.fromV].edgeexists
    THEN BEGIN
      son[e.toV].markedge;
      IF NOT queued
      THEN BEGIN
        parent.insertInQ(qe);
        queued:=TRUE;
      END;
    END;
  END of procedure load;

  queued:=false;
  qe:=NEW queueElement(THIS vertex);
  BEGIN INTEGER i;
    FOR i:=1 STEP 1 UNTIL n DO son[i]:=new toVertex(i);
  END of init code;
END of class vertex;

```

Figure 11. Revised Warshall Implementation (Continued on next page)

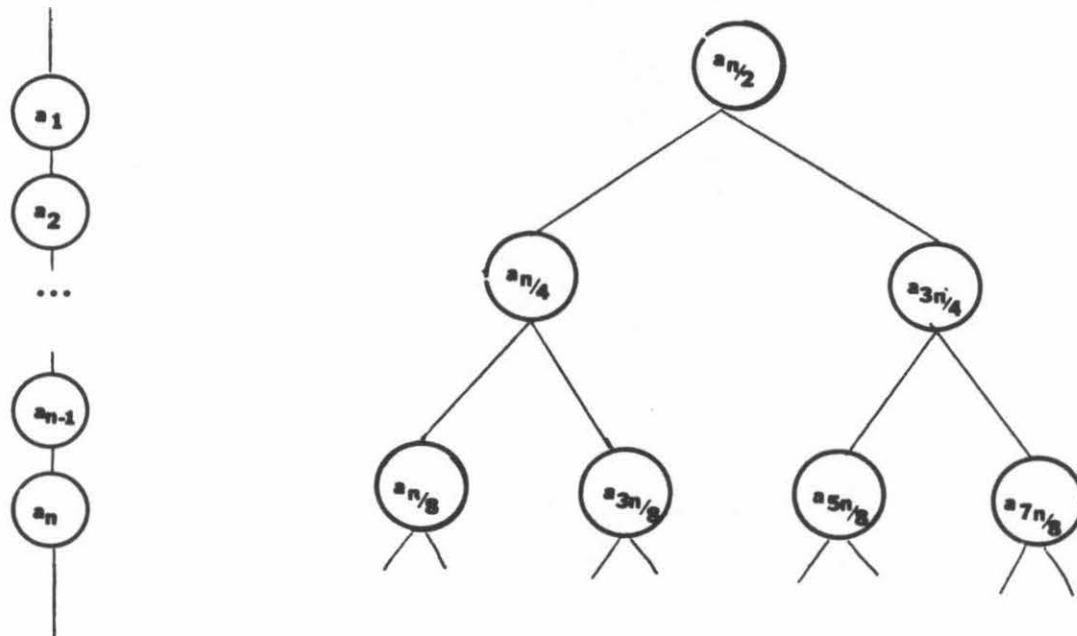
```

Qprocessor CLASS toVertex(myNode); INTEGER myNode;
BEGIN
  REF(queueElement)qe;
  BOOLEAN edgeExists;

  PROCEDURE markEdge;
  BEGIN
    IF NOT edgeExists
    THEN BEGIN
      edgeExists:=TRUE;
      parent.insertInQ(qe);
    END;
  END of procedure markEdge;

  edgeExists:=FALSE;
  qe:=NEW queueElement(THIS toVertex);
END of class edge;

```

Figure 11. Revised Warshall Algorithm Implementation**Figure 12. Arrangements of s_i in the Tree and Ringmachine**

the transitive closure.

Each arc in the original graph is given to the root via a call on procedure load. The arc is passed to all vertex processors. There, on the second level, each vertex executes the test described above to see if the arc causes the creation of an arc from this vertex.

Once all the arcs of the original graph have been loaded, the arcs of the transitive closure are available for unloading. As an arc is handed to the outside world by a call on the root's unload procedure, it is passed back down the tree to the vertex processors, just as the original arcs were, by a call on procedure load.

A double system of queues is used to indicate the availability of arcs for the unloading and broadcasting operations. The queue in the root is used by the vertex processors to indicate willingness to provide an arc to the root. When an arc is unloaded, it is also broadcast through the tree via the load routine. The queue in the vertex processors is used by the toVertex processors to indicate that another arc has been created.

The queues are used to avoid polling the vertexs and toVertexs from available arcs. The polling introduces two iteration statements which are executed for each arc in the transitive closure. They cloud the issue by appearing to affect the complexity. The queues, on the other hand, simulate the hardware nicely. The two upper levels of the tree need to respond to a signal from any one of their children. The queues provide this effect.

The algorithm as described above and in Figure 11 has time complexity of the order of the number of arcs in the transitive closure. The maximum number of arcs in a directed graph of size n , is n^2 ; the transitive closure is itself such a graph, is n^2 . Thus the time complexity of this algorithm is $O(n^2)$, limited by the time it takes to read out the arcs of the closure.

As described, $2n^2 - 1$ processors are used to generate the closure. A solution using only $n+1$ processors, yet essentially the same, can be devised. Suppose each vertex processor, now the leaves of the tree, contains a boolean array instead of using toVertex processors to represent existing arcs. The vertex processors have more local store, and a parameter of the problem, the size of the graph, has been introduced into the physical requirements for each processor. This is something I want to avoid. It is, however, a perfectly valid implementation, and indeed, retains the $O(n^3)$ total complexity.

Is the Tree Machine Magic?

It is time to address the question of whether these problems need the tree machine structure. The answer is simple. No. I will give an alternative architecture that yields an equivalent solution.

Mike Ullner has proposed the Ringmachine [7], a "tree of branching ratio one". The structure is a doubly-linked ring of processors, or more simply, a linear pipeline.

This structure is also capable of doing transitive closure in $O(n^2)$ time using $O(n^2)$ processors, and the code is as simple as that for the tree machine implementation. The Ringmachine algorithm is given in detail in [3].

The key to the $O(n^2)$ solution is the pipeline, not the communication path. In fact, sorting and matrix multiplication are also problems in this class. The size of the answer determines the size of the problem. Any pipelined structure that can spew out answers one at a time in a continuous stream is adequate.

So what is the tree machine better at? The difference between the tree and the ring is that any particular node in the tree can be total number of processors. Problems that have one answer that can be in any of a large number of processors can take advantage of the

tree structure. NP-complete problems, like the color cost problem treated earlier, are a graphic example of this. Those problems require an exponential number of processors, however, and thus are not practical.

An Algorithm that Uses the Tree Effectively.

I have found a problem that does make use of the extra communication paths in the tree. It is taken from numerical analysis, and is presented here out of context. The problem is to generate a vector x from a vector a according to the following rule:

$$x_i = \sum_{j=1}^i a_j$$

In other words, the i th element of the vector x is the sum of the first i elements of the vector a . This problem is solvable on a sequential machine in $O(n)$ time.

If the tree machine and Ringmachine are treated as peripheral functional units that are given a and produce x , the performance of the two machine is identical. Loading and unloading the vectors again dominates the time complexity. In each case, n processors are used to solve the problem in $O(n)$ time.

A more interesting formulation of the problem assumes that the tree and Ringmachine are already loaded with some convenient arrangement of a . How fast can x be generated, with x ending up in the same arrangement as a ?

Given the arrangements shown in Figure 12, the Ringmachine uses n processors to generate x in place in $O(n)$ steps. The tree machine, on the other hand, uses n processors, but arrives at the answer in $O(\log_2 n)$ steps. For large n , this is a significant difference.

```

CLASS sum(s,max); INTEGER s,max;
BEGIN END;

Processor CLASS vectorSum;
BEGIN
  INTEGER subscript;
  INTEGER x;

  REF(sum) PROCEDURE sumup;
  BEGIN
    IF left==NONE AND right==NONE
    THEN sumup:=NEW sum(x,subscript)
    ELSE BEGIN REF(sum)l,r;
      l:=IF left==NONE THEN NEW sum(x,subscript) ELSE left.sumup;
      r:=IF right==NONE THEN NEW sum(x,subscript) ELSE right.sumup;
      x:=x+l.s;
      sumup:=NEW sum(x+r.s,r.max);
      left.sumdown(l);
      right.sumdown(NEW sum(x,subscript));
    END;
  END of procedure sumup;

  PROCEDURE sumdown(p); REF(sum)p;
  BEGIN
    IF p.max<subscript THEN x:=x+p.s;
    IF left=/=NONE THEN left.sumdown(p);
    IF right=/=NONE THEN right.sumdown(NEW sum(x,subscript));
  END of procedure sumdown;
END of class vectorSum;

```

Figure 13. Algorithm for finding x_i .

	Sequential Machine		Tree Machine	
	space	time	processors	time
Heap Sort	n	$n \log_2 n$	n	n
Matrix Multiplication	n^2	$\sim n^3$	n^2	n^2
Color Cost	n	n^n	n^n	n^n
Transitive Closure	n^2	n^3	n^2	n^2
x_i	n	n	n	$\log_2 n$

Figure 14. Sequential and Tree Machine Performance.

The Ringmachine algorithm is straightforward. Starting with the vector a distributed as in Figure 12, each processor adds numbers that are passed in from the left to the a_i it holds before passing them on. After the i th processor has seen $i-1$ numbers, it sends a_i to the right and becomes dormant. The n th processor waits $n-1$ time steps for a_1 . The other $n-2$ elements arrive in the next $n-2$ time steps, and are added to a_n to form x_n . Thus the process is complete after $2n-1$ cycles.

The algorithm on the tree machine is not as simple. The arrangement of the a_i 's given in Figure 12 is not intuitive. And the algorithm requires data to flow up and down the tree simultaneously. The SIMULA code is given in Figure 13.

The summing starts in the lower left hand corner of the tree. Each node gets partial sums from its left and right children. The left hand sum is added to the a_i in the processors, stored as x_i , and passed to the right child. Then the sum from the right child is added in, and this result, the sum of all three numbers available, is sent to the parent processor. It takes $\log_2 n$ cycles for the root to receive the sum of the a_i 's in the left half of the tree, and another $\log_2 n$ steps for that sum to filter down to the lower right corner, forming x_n .

The algorithm described above uses the extra communication paths of the tree to advantage. It remains to be seen if the problem can be put back into the numerical analysis context from which it came, and still perform better on the tree than on the Ringmachine.

Conclusions.

The work described in this paper is aimed at deciding two questions. First, are multiprocessor systems useful? And if so, what kind of system should be built?

The answer to the first question is a resounding yes. Figure 14

compares the performance of the algorithms described here on sequential machines and the tree machine. In each case, the time complexity is substantially reduced.

The second question does not yet have a clear answer. I am just beginning to examine problems that can use the three-neighbourness of the tree to advantage. Unless the additional complexity of building a tree rather than a Ringmachine can be justified, the simpler structure is heavily favored. I am hopeful, however, that numerical analysis problems will demonstrate the value of the tree machine.

References

- [1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman
The Design and Analysis of Computer Algorithms
Addison Wesley, Reading, Massachusetts, 1974
- [2] Birtwistle, G.M., O-J Dahl, B. Myhrhaug, K. Nygaard
SIMULA BEGIN
Petrocelli, New York, 1973
- [3] Browning, Sally A.
"Transitive Closure and the Tree Machine"
Computer Science Department Display file #2402
California Institute of Technology, 1978
- [4] Dahl, O-J, E.W. Dijkstra, C.A.R. Hoare
Structured Programming
Academic Press, New York, 1972
- [5] Demuth, H.B.
"Electronic Data Sorting"
PhD. Thesis (Stanford University, October 1956)
- [6] Dijkstra, E.W.
A Discipline of Programming
Prentice-Hall, Englewood Cliffs, New Jersey, 1976
- [7] Ullner, Mike
"Ringmachine"
Computer Science Department Display file in progress
California Institute of Technology, 1978
- [8] Marshall, S.
"A Theorem on Boolean Matrices"
J.ACM 9:1, p.11-12

A DATA-DRIVEN MACHINE ARCHITECTURE
SUITABLE FOR VLSI IMPLEMENTATION

A. L. Davis
Computer Science Department
University of Utah
Salt Lake City, Utah 84112

ABSTRACT:

A machine architecture is presented which is capable of supporting very high levels of concurrency. The machine language of the class of machines described here is a graphical program schema known as data-driven nets. The machine architecture is arbitrarily extensible and consists of a recursively organized hierarchy of homogeneous processor-store modules. System control is decentralized, and each module is a completely asynchronous processing site, capable of executing any machine language program. Resource allocation is performed dynamically on the basis of the amount of available concurrency in the program and on the availability of physical resources.

Key Words: VLSI, concurrency, pipelining, recursive hierarchy, data-flow.

I. INTRODUCTION

In an attempt to increase the performance of computing machines, there appears to be two main approaches: 1) to use faster components in existing architectures, and 2) to design new architectures and programming methods, which are capable of exploiting high degrees of concurrency. The first approach is inherently limited in that the effects of reduced integrated circuit geometry, and new logic families can be expected to increase overall system performance by only a couple of orders of magnitude. While this is initially impressive, it does not meet the desired machine performance estimates necessary to solve large physics problems, or that needed for accurate weather prediction [16]. The second approach is not inherently limited by the physical properties of switching devices. There are numerous levels at which concurrency can be exploited in digital computers, i.e. multiple data paths, more concurrent realization of low-level circuit functions, overlap and pipeline processing within a single processing element, multiple processors, etc. In developing any new "fast as possible" machine, it is important to attempt to implement all of the above suggestions. The work reported here will mainly be concerned with solving the problem of how to utilize and organize systems containing large numbers of independent processors.

In attempting to escape the performance bounds imposed by Von Neumann architectures, it is insufficient to modify only a few aspects of the Von Neumann style system ideas. Alternative proposals to the "clock-driven" Von Neumann architectures are numerous. There are two areas which have some promise. One is the "demand-driven" approach espoused by Friedman and Wise [10]; Backus [3]; and Berkling [5]. Another is the "data-driven" approach taken by Dennis [8]; Bahrs [4]; Davis [6]; and Arvind, Gostelow, and Plouffe [2]. The work described here is of the data-driven variety due to the difficulty with which demand-driven systems support intra-process pipelining. In addition the propagation of demands takes time, and while demand-driven programs do allow for increased expressive power, the emphasis here is on performance. The data-driven approach naturally describes both pipelined (vertical) concurrency, and independent operation (horizontal) concurrency.

The work reported in this paper was supported by Burroughs Corporation. DDM1, a prototype module of the architecture described in this paper, is a hard wired data-driven machine, and was completed in July 1976 at the Burroughs Interactive Research Center in La Jolla, California. Many of the early systems ideas were developed in conjunction with Robert S. Barton. Gary Hodgman, Lawrence Rogers, and Karl Boekelheide were instrumental in the conceptualization and implementation of the actual machine. DDM1 now resides at the University of Utah, where the project continues under the support of the Burroughs Corporation.

It is clear that any machine architecture intended to have a general commercial appeal must be viable with respect to the changing constraints of integrated circuit technology. For architectures which fit nicely into the VLSI realm, the advantages are numerous. Among these are lower cost, increased reliability, increased speed, and decreased power consumption.

The actual machine language of the class of machines presented here is a directed graph schema called data-driven nets [6] or DDN's. DDN's are very similar to the data-flow nets of Dennis [8] and Rodriguez [15]. The terms data-driven and data-flow are used synonymously. The asynchronous nature of DDN's makes it easy to decompose a given net into a set of concurrent subnets, which can then be allocated to independent physical resources. The main distinction between the Dennis nets and DDN's is that in DDN's no distinction is made between the net tokens which are used for control purposes, and other net tokens. All DDN tokens are considered to be data items, and no explicit distinction is made to distinguish between classes of tokens. Another difference is that the primitive DDN cell types are slightly more high level than the Dennis nets. Finally some primitive activities which are explicitly specified in the Dennis schema are implicit in DDN's. An example is the Dennis "link" which serves as both a transmission and copy site. The functions of such links are implicitly incorporated into the output mechanism of DDN operators. The result is that both the DDN and Dennis schemas share the same properties with respect to ease of program verification, ease of program conceptualization, and ease of machine evaluation. Due to slightly higher level primitives and a less explicit schema a DDN program graph will typically have less vertices (cells) and arcs (data paths) than a functionally equivalent net in the Dennis schema. This difference is mostly one of style and is not particularly significant, although the differences are reflected in the respective architectures.

The only sequencing constraint in DDN's is that of data dependence, and since no weaker sequencing constraint exists without doing non-productive computation [14], DDN's are naturally a maximally concurrent representation of a given algorithm. While such concurrency may add to the "naturalness" of the programming experience, it is useless as a speed-up mechanism unless it can be mapped onto a set of physical resources capable of exploiting this concurrency. If this mapping is done at run-time, then the time to map must not overshadow the speed-up attained as a result of the concurrent execution.

Lastly, a number of additional goals for the machine structures presented here are felt to be desirable. Namely it is intended that these machines be general purpose, extensible, reliable, easily programmable, support very high levels of concurrency, and also be economical with respect to their performance and existing technology. In particular this effort is not concerned with one of a kind or special purpose machines. Special

purpose machines are perhaps ideal for a given environment, but suffer from inherent limits in their applicability to other problems.

II. THE IMPACT OF VLSI

The advantages of high density integrated circuit technology are so overwhelming that the constraints of VLSI must be considered as a primary force on future architectures. A detailed analysis of these effects is beyond the scope of this paper, but the global influences are summarized here. Due to the tremendous commercial emphasis on MOS VLSI, the following discussion will mainly be concerned with the properties of MOS device integration.

The most highly publicized VLSI benefits are those involving cost. A single custom VLSI chip (64 pin package) currently costs about \$80,000 to \$300,000 to produce. Even then, production typically must be guaranteed for about a quarter of a million parts at an additional cost of \$7 to \$10 per part. This clearly indicates that VLSI cost advantages can be obtained only if any given chip can be used in very large volumes. If a part does not have universal appeal, then the use of such a part in a new architecture brings about some high pressure constraints. Either the part must be used a large number of times in a single system, or a single system must have a very high sales volume, or some combination of the two. The number of part types in a given system is also a major concern in that it becomes a multiplicative factor in the system development cost.

Another factor heavily influenced by a VLSI implementation is speed. The dominant speed factor is due to the capacitive effects on a given transmission path. Typical off chip loads are on the order of 100 picofarads, while on chip loads are approximately one picofarad. Since delay times are proportional to the capacitive load (for constant drive current), this implies that signals which can remain on the chip will be driven about 2 orders of magnitude faster than those which must be driven to destinations off the chip. Additional speed-up can be obtained from the decreased geometries of switching elements and conductor path lengths. This is a very strong argument for architectures which attempt to maximize locality of processing. For architectures in which processing and local storage can not be done at the same locality, massive off chip delays must be incurred as a result. The only way around the slow off chip drive problem is to drive more current off the chip. This requires a series of relatively large output drivers, which are extremely costly in terms of chip real estate and power consumption. In addition, locality of processing will reduce the amount of contention for a given system transmission path. This contention is important in a highly parallel system in that the resultant sequencing will yield reduced system efficiency.

The number of pins is an important VLSI metric. The pin count is a primary factor in determining whether a given system module is nicely implementable as a VLSI circuit. Techniques to decrease physical pin count, such as time division multiplexing are applicable in certain situations but can not be considered a general solution. In addition, if chip types are used in sufficient quantities to amortize the initial layout cost, then the physical cost to manufacture the system becomes approximately linear with pin count. In addition increasing the number of pins on a particular chip causes decreased yield due to bonding problems. Increased pin count also implies that even more silicon area must be allocated to connection pads and pin drivers.

VLSI implementation also yields the more commonly discussed advantages such as: 1) increased system reliability due to reduced part count, 2) decreased system power consumption since voltages on a given chip scale with physical feature size, and 3) decreased system maintenance cost as chip replacement policies become more effective in highly integrated systems.

The extent to which these VLSI advantages can be realized is proportional to the logic/pin ratio of the proposed system modules. If the logic/pin ratio is relatively small then the situation is very much that of an SSI machine. If the logic/pin ratio is very high then true VLSI advantages can be obtained. This is a challenge to architects to devise systems which can be modularized into high complexity modules which communicate with their environment using relatively few signals. Furthermore as integration technology advances causing feature sizes to shrink even more, these new architectures must remain viable.

III. ARCHITECTURAL PRINCIPLES

The VLSI constraints indicate that future architectures to support very high levels of concurrency should consist of a set of processing sites capable of performing localized storage and computation of a reasonable complexity. These sites should be essentially the same physical module, which can be constructed from one or a set of part types. An additional goal of the architecture presented here is that of extensibility. More specifically, the architecture should be extensible without bound in the following way:

1. Machine power should be enhanced by the addition of more modules (i.e. allow greater concurrency due to the increased number of processing sites);

2. The addition of new modules should not require any change to the existing operating system in order to manage the resulting larger system;
3. Additional resources should be added simply by "plugging in new modules" without any special tuning of the existing hardware to create consistent system timing and communication for the expanding system; and
4. Extensions should be available in small quantum.

The first and last points indicate that a user should be able to purchase only the power needed and not much more or much less. The other points indicate that the manufacturer only needs to support a single module, rather than a large number of system configurations.

Systems such as these cannot be implemented in a synchronous, centrally controlled manner. Central control of arbitrarily extensible systems implies that the control must be able to function on an arbitrarily large amount of state information, which either slows the control drastically or requires controller modification to access the new state information. In an arbitrarily extensible synchronous system the problem of unbounded clock skew (maximum difference in the perceived clock time between any two processing sites in the system) will cause failure. The systems described here will therefore be asynchronous, fully distributed systems. Fully distributed systems have the following characteristics: 1) no module of a fully distributed system can determine the total system state, and 2) no module of a fully distributed system can enforce simultaneity in other modules. Holt [13] has shown that the notion of total system state in complex asynchronous systems is counter productive. Furthermore the enforcement of simultaneity in physically separate, asynchronous devices is impossible.

There are many ways to organize an extensible set of modules in a distributed control system. The advantages of hierarchical organizations are: 1) simplification in the amount of complexity to be dealt with at a given level, 2) verification by inductive methods can be done for uniform hierarchic systems, and 3) the superior-inferior relationship can be utilized to resolve problems such as contention and deadlock in multi-resource systems. It will be seen that hierarchy also facilitates a nice resource allocation policy. Recursive hierarchies are of particular interest in that they imply that the same module (and ultimately the same chip) can be used at each level.

Recursive systems are nicely extensible. A recursively structured machine is one which has exactly the same structure at every level. Clearly physical recursion must terminate at some point. This point will be seen to be the deepest set of resources in the physical hierarchy. Additional advantages of recursively structured systems have been demonstrated by Glushkov[11]. It will be shown that the width of a level in these recursive hierarchic structures can be used to execute independent operations, while the depth of the hierarchy will be used to facilitate pipelined operations.

IV. THE ARCHITECTURE

The architecture consists of a set of asynchronous modules which communicate by passing messages. The basic computational unit of the architecture is a processor-store element (PSE). A PSE consists of a processor module (P) and its associated local storage module (S). Any PSE can execute any machine language program, providing that it has a sufficient amount of local storage. No module that is not a PSE can perform this function. The architecture is a recursively organized set of these PSE's. The recursive definition of the structure is:

$$\begin{aligned} \langle \text{PSE}_n \rangle &::= \langle \text{P}_n \rangle \langle \text{S}_n \rangle \\ \langle \text{S}_n \rangle &::= \langle \text{ASU}_n \rangle \\ \langle \text{P}_n \rangle &::= \langle \text{AP}_n \rangle \mid \langle \text{AP}_n \rangle \langle \text{PSE.GROUP}_{n+1} \rangle \\ \langle \text{PSE.GROUP}_{n+1} \rangle &::= \langle \text{PSE}_{n+1} \rangle \mid \langle \text{PSE}_{n+1} \rangle \langle \text{PSE.GROUP}_{n+1} \rangle \end{aligned}$$

Subscripts denote the recursive level at which the module physically resides. $\langle \text{AP} \rangle$ is an atomic processor module, which has no further sub-structure (contains no PSE's). Similarly an atomic storage unit $\langle \text{ASU} \rangle$ has no PSE substructure. The width of a $\langle \text{PSE.GROUP} \rangle$ has a physical bound. For the DDM1 prototype, this bound is eight. The structure is depicted in Figure 1.

This structure allows for a hierarchical distributed storage organization. Any S or ASU may consist of an arbitrary amount of storage of any desired medium. Higher levels of PSE's are considered logically superior to lower level PSE's. It is advantageous if higher level stores (ASU's) are slower and larger than the stores of lower levels. The interface and functional ability of any ASU (regardless of size, speed, and level) is the same. The structure also allows for an arbitrary number of processors to be used concurrently. It is important to note that all AP's are identical regardless of level. However, the processors at higher levels

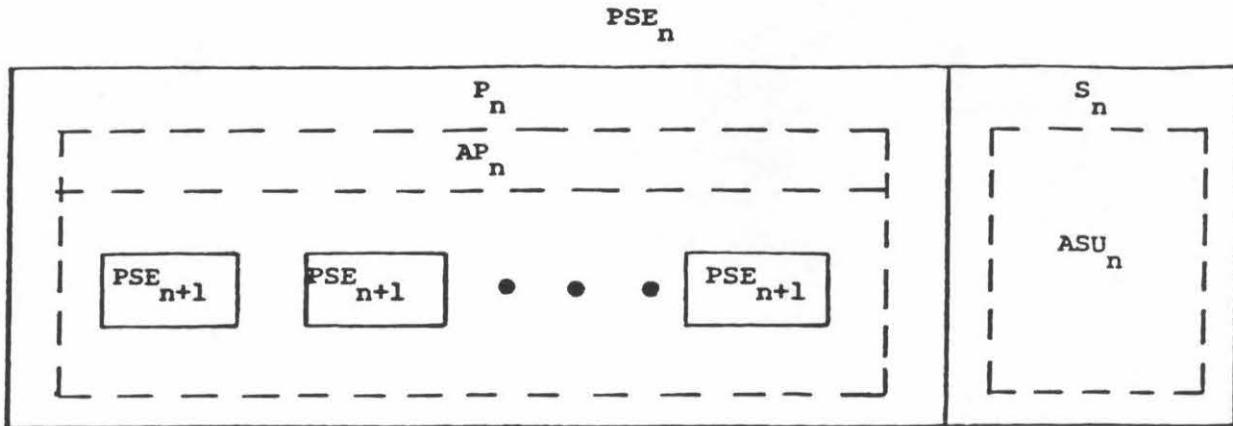


Figure 1: Recursive definition of PSE at level n .

will be more powerful, in that they contain more substructure than the processors at lower levels. More substructure implies more internal concurrent processing capability.

When viewed non-recursively this structure is simply a tree structure with a single root and a possibility for up to eight sons at any node. Each node of the tree is a PSE and is capable of executing any machine language program. The leaf nodes have no substructure and therefore consist of an AP and an ASU. At each node the fan-out is fixed but the depth of the tree is arbitrary. In this manner the architecture allows any desired number of PSE's to be configured for a given machine. The desired goal is for machine performance to improve with the addition of more PSE's.

There are a number of ways to enforce this logical tree structure onto a collection of PSE's. All involve some form of a connection network to implement the desired communication paths. A number of general interconnect networks have been considered: busses, crossbars, Banyan nets [12], and

permutation networks [17]. For tree-like machines, full connectivity is not required. The expense of crossbar switches vary as the square of the connected elements. Bus conflict would drastically reduce actual parallelism in the machine. Permutation networks present a tremendous problem in that they may need to be totally reconfigured when a single new connection is necessary. This is difficult to do reliably in a multi-path distributed control environment. Banyan networks have some merit, but do not easily allow for the desired hierachic pipelined communication. Therefore in the DDM1 prototype, a simple 1 to 8 switch was chosen as the interface unit between successive levels of PSE's. The result is that the physical and logical recursive structures are the same. The structure is fixed and cannot be dynamically changed.

Information is passed between PSE's as messages which are variable length character strings. Upward traveling messages are passed on by the switch in an arbiter like manner. Downward going messages contain header fields which indicate their destination. This header is deleted by the switch as the message is passed. Downward and upward messages are dealt with by independent hardware, and therefore are controlled concurrently. This character serial nature of the machine has the following advantages:

1. Hardware modules are made simpler and more applicable for VLSI implementation due to the reduced pin count.
2. Hardware communication paths are more general in that variable length information units can be transmitted as varying numbers of fixed-width base characters. This facilitates a hardware substitution strategy for modules. Each module can interpret the variable length message and perform the indicated function.

These advantages aid in greatly reducing the cost of the hardware modules. Some low-level performance is lost by doing everything serially. The philosophy for this architecture is to regain that lost performance many times over by providing a systems organization that allows for many highly concurrent levels of activity.

Physical queues are placed between levels of PSE's in order to facilitate pipelining and increase physical module independence. Without queues, the sender of a message would need to wait on receiver availability. If a queue becomes full, only then must the sender wait until the receiver has freed up some queue space. If queue sizes are adjusted so that a sender is rarely required to wait for space, then the system would be well tuned for efficient processing. Optimal queue size depends on the average message length. It is therefore impossible to guarantee that no waiting will occur.

Strict hierarchical control and a restricted process structure insures that the system does not deadlock. A block diagram of the PSE structure is shown in Figure 2. In the DDM1 prototype, all communication paths except for the path between the ASU and the AP, consist of 6 wires (a 2 wire request-acknowledge control link and a 4 wire, character-width data bus).

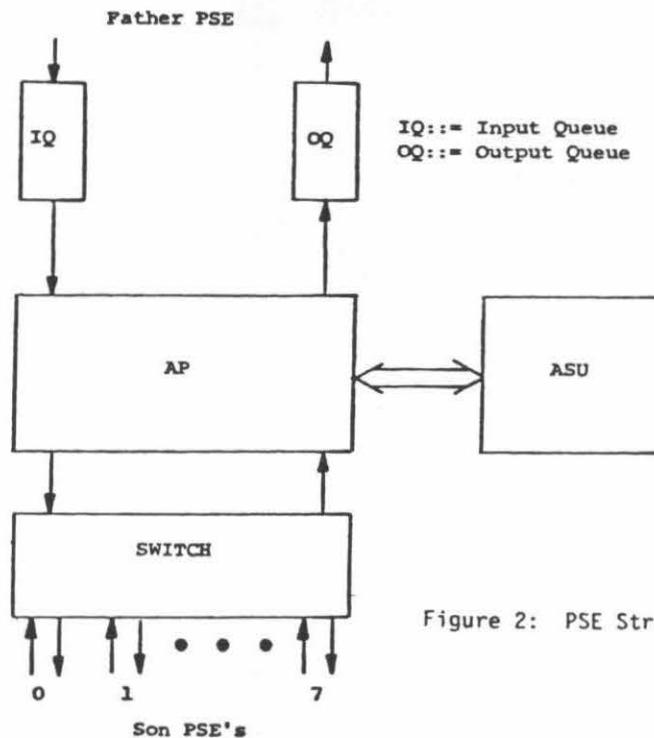


Figure 2: PSE Structure

The variable length, character serial message structure and DDN representation indicate that the ASU should be a highly flexible storage structure. Further requirements are that the ASU deal with the pipelining of data items and the continual destruction of data items due to cell firings. In order to increase efficiency of the PSE, all storage management functions are performed internally by the ASU. The ASU appears as a variable field length file system, which directly executes commands, such as: initialize, skip, insert, read, write, delete, and index. The free space is managed automatically by the ASU.

This PSE structure allows for a high degree of processing locality in that any PSE can execute any DDN program (assuming that there is sufficient storage in its local ASU). In addition the PSE admits nicely to VLSI implementation. The 1 to 8 switch can be implemented using a set of 1 to 2 switches of similar function. Using 1:2 switches, module complexities for the DDM1 prototype (pin and gate count) are shown in Figure 3. The pin counts include pins for power, ground, initialization, and extension. The indicated module pin counts are rounded up to coincide with standard package sizes.

<u>Module</u>	<u>Gate Count</u>	<u>Pin Count</u>
IQ, OQ (1K Characters)	3,000	16
AP	20,000	64
ASU (4K Characters)	47,000	64
1:2 Switch	2,000	40
Ap + ASU	67,000	64
Ap + ASU + IQ + OQ	73,000	64
Ap + ASU + Switch	69,000	64
PSE	75,000	64

Figure 3: PSE Module Complexities

These complexities are all within reason for VLSI designs, and are attractive with respect to the logic/pin ratio.

V. AUTOMATIC RESOURCE ALLOCATION AND EVALUATION

When a message corresponding to a DDN program enters a PSE at any level, the PSE may take one of two actions:

1. DECOMPOSITION AND ALLOCATION: if the PSE has substructure and if there exists some set of concurrent subnets in the DDN process, then the PSE may split the DDN and send concurrent subnets to PSE's at the next lower level.
2. EXECUTION: if the PSE has no subresources, or if there is no exploitable concurrency in the DDN, then the PSE executes the DDN at that level.

To aid the decomposition process, a structural descriptor may precede the incoming DDN in the message structure. This additional storage can greatly reduce time required for decomposition decisions in the PSE. In addition, each PSE must contain information about the number of available PSE's and the sizes of their respective stores. Problems would result if a DDN were sent to a PSE that was too large to fit in its local store. Only the local store sizes of immediate subresources are known. This insures the recursive nature of the decomposition process.

The decomposition process takes some time. It is important that the speed-up gained by extra concurrency resulting from the decomposition is not overshadowed by the time to decompose. Experiments have indicated that a "first fit" decomposition is almost always better than a "best fit" decomposition strategy. It is also not generally worthwhile to decompose the DDN structure completely on this architecture. At fine granularities, the slowdown resulting from loss of locality is not regained by the concurrent execution of very small subtasks. The exception to this rule is in the case of pipelining, where subtasks remain allocated for relatively long periods of time and sustain high activity at each site.

If decomposition and resource allocation occur at run-time, it is important that they be simplified as much as possible. It is possible to perform these tasks completely at compile-time. This however is inadvisable since it depends on knowing the run-time availability of PSE's in the system. In a system containing large numbers of PSE's, the probability is high that some PSE's will fail or be busy doing other things. In addition large portions of a process may only be evaluated conditionally. A compile-time allocation would have to allocate tasks which may never be executed. The strategy is taken here to split the decomposition task into two phases: 1) at compile time do all of the resource and condition

independent work, and 2) at run-time, dynamically make the actual allocation of executable tasks to available physical resources.

DDN's are quite randomly structured graphs and the data-driven architecture is a very regularly structured set of resources. Direct run-time allocation would be too slow, due to the structural disparity between program and machine. At compile-time, the two-terminal DDN process structure is transformed into a well structured and functionally equivalent series parallel graph (SP-graph). Two-terminal means that the graph contains a single "first" cell and a single "last" cell. This facilitates the determination of net termination and initiation. SP-graphs are acyclic, two terminal, directed graph structures which can be formed by successively combining cells and/or SP-graphs in series or in parallel. The SP-graph structures are then allocated as necessary at run-time. Data-flow graphs in general admit nicely to arbitrary restructuring due to their asynchronous and local control characteristics.

VI. CONCLUSIONS

An architecture and evaluation scheme for data-flow programs has been presented. The architecture exploits recursive hierarchy to reduce complexity and allows for the arbitrary expansion of system resources. Physical resources are organized such that they can be used to exploit both pipelined and independent tasks. The system exploits the notion of locality that is important for both increased speed and decreased cost aspects of a VLSI implementation. This notion of locality also indicates that this system is not intended to exploit concurrency at the lowest possible level. It is felt that the additional overhead involved to do this would actually reduce overall performance levels.

The main points of departure of this approach and that of Dennis [9] is the use of a recursive hierarchy of physical resources, the exploitation of physical locality to decrease message frequency and increase the speed of VLSI implementations, dynamic hierarchical resource allocation, the lack of specialized functional modules to reduce the chip type count, and a slight difference in the structure of the low-level schema. The architecture of DDM1 differs from that of Arvind and Gostelow [1] in that it does not try to achieve concurrency at all possible levels (because of the locality issue), the interconnection scheme is much simpler which results in reduced communication path contention, no special address space management needs to be done, allocated tasks may consist of many cells rather than just a single operation, and tasks are allocated only when all of their necessary input operands are present.

Analysis based on a working prototype module indicates that the machine architecture described is nicely implemented in VLSI. It has been shown that it is possible to implement an entire processor-store element as a 64 pin chip containing about a quarter of a million MOS transistors. This logic/pin ratio indicates that true VLSI benefits can be obtained.

The disadvantages of the system described here are:

1. The current ASU design is not nicely extensible to allow more storage capacity to just be "plugged in".
2. The fixed, hard-wire tree structure is not reconfigurable and may result in a situation where certain PSE's in one subtree remain idle when another heavily loaded subtree badly needs more resources.
3. There is currently not enough empirical data from test runs on very large programs to accurately quantify the overhead involved in decomposition and resource allocation.

BIBLIOGRAPHY

Arvind, Gostelow, and Plouffe. The ID Report: An Asynchronous Programming Language and Computing Machine. University of California at Irvine, Computer Science Department, Technical Report #114, (1978).

Arvind, and K. P. Gostelow. A computer capable of exchanging processors for time. Information Processing '77, North Holland, New York (1977), pp. 849 - 854.

Backus, J.. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM, Vol. 21, No. 8, pp. 613 - 614, (August 1978).

Bahrs, A.. Programming language semantics and closed applicative languages. Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 71 - 86, (1972).

Berkling, K. J.. Reduction Languages for Reduction Machines. Interner Bericht ISF-76-8, GMD, (1977).

Davis, A.. The Architecture of DDM1: A Recursively Structured Data-Driven Machine. University of Utah, Computer Science Department, Technical Report UUCS-77-113, (1977).

Davis, A.. Data-Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems. University of Utah, Computer Science Department, Technical Report UUCS-78-108, (1978).

Dennis, J. B.. Data Flow Schemas. Lecture Notes in Computer Science 5, Springer-Verlag, pp. 187 - 216, (1972).

Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM National Conference, pp. 402 - 409, (1974).

Friedman, D. P., and Wise, D. S.. Aspects of Applicative Programming for Parallel Processing. IEEE TC, Vol. C-27, No. 4, pp. 289 - 296, (April 1978).

Glushkov, V. M., et al. Recursive Machines and Computing Technology. IFIPS Proceedings 1974, North Holland, New York, pp. 65 - 70, (1974).

Goke, L. R.. Banyan Networks for Partitioning Multiprocessor Systems.
Ph.D. Dissertation, University of Florida (1976).

Holt, A., and F. Commoner.. Events and Conditions. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, pp. 3 - 52, (1970).

Linderman, J. P.. Productivity in Parallel Computation Schemata. MIT Project MAC, TR-111, (1973).

Rodriguez, J. D.. A Graph Model for Parallel Computations. MIT Project MAC, TR-64, (1969).

Rudy, T. E.. Megaflops from Multiprocessors? Proceedings of the 2nd Rocky Mountain Symposium on Microcomputers, pp. 99 - 107, (1978).

Waksman, A.. A Permutation Network. JACM, Vol. 15, No. 1, pp. 159 - 163, (1968).

Area-Time Complexity for VLSI

C. D. Thompson
Carnegie-Mellon University
Pittsburgh, Pennsylvania

Abstract

The complexity of the Discrete Fourier Transform (DFT) is studied with respect to a new model of computation appropriate to VLSI technology. This model focuses on two key parameters, the amount of silicon area and time required to implement a DFT on a single chip. Lower bounds on area (A) and time (T) are related to the number of points (N) in the DFT: $AT^2 > N^2/16$. This inequality holds for any chip design based on any algorithm, and is nearly tight when $T = \Theta(N^{1/2})$ or $T = \Theta(\log N)$.

1. Introduction

The theory of computation is valid over a synthetic domain: its formal models have relevance only if they correspond to possible computational systems. Technological changes can affect the realm of possibility. In this light, it would be surprising if the "VLSI revolution" did not spawn new theoretical models. This paper is an attempt to show that interesting complexity results are available through the use of a "VLSI model of computation".

Two parameters are of overriding interest in a VLSI design, its speed and its size. Speed can be handled with familiar complexity tools, that is, measuring time by counting elementary operations. Size in the VLSI world is best expressed as the total area of silicon used. This is quite a different metric from a count of "active elements", "gates", or "registers". It may be the case that most of the chip is devoted to connections between such active elements. A complexity theory for VLSI must thus concern itself with the layout of active elements in the plane, along with their interconnections.

This research is supported in part by the National Science Foundation under Grant MCS 75-222-55 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422. C. D. Thompson is an NSF fellow.

The VLSI model: area and time.

There is a natural unit of area for VLSI. Manufacturing and physical limitations give rise to a "minimum feature width", λ . This is the width of the narrowest wire, and λ^2 is approximately the area of the smallest transistor. The 64K RAM currently available has an area of about $10^5 \lambda^2$. Chips of 10^7 or $10^8 \lambda^2$ may be possible [Mead 78].

The choice of a unit of time is slightly more problematical. Here, following [Mead 78], it will be taken as the length of time that it takes a signal to propagate along a wire, or on-chip interconnection. This propagation time can be made independent of the length of the wire, by fitting larger drivers to longer wires. Larger drivers of course occupy more area, but need never take more than 10% of the area of the wire they drive ($1\lambda^2$ for a wire of length 10λ , $10^4\lambda^2$ for a $10^5\lambda$ wire). By fudging λ upwards by 5%, the area of the driver is thus absorbed into the area of its wire.

A full exposition of the VLSI model is deferred to Section 2.

The DFT.

The computational problem studied in this paper is the Discrete Fourier Transform (DFT). The DFT is defined over any commutative ring, but only finite rings will be considered here. Elements of infinite rings have no fixed-length representation, leading to grave computational difficulties. Approximate methods are beyond the scope of this paper.

A satisfactory ring does exist for VLSI, the integers modulo m . If $m = 2^k - 1$, ordinary fixed-point arithmetic on k bit words will produce exact answers. An N -point DFT can be performed in this ring if N divides $p-1$ for each prime p dividing m [Bonneau 73].

Formally, the DFT is a matrix-vector multiplication, $A\tilde{x} = \tilde{y}$. The input vector is \tilde{x} , the output vector is \tilde{y} , and A is an N by N matrix of constants,

$$A[i,j] = w^{ij}$$

The constant w must be a principal N th root of unity. That is, it must satisfy

$$\begin{aligned} w &\neq 1, \\ w^N &= 1, \text{ and} \\ \sum_{0 \leq j < N} w^{jp} &= 0, \text{ for } 1 \leq p < N \end{aligned}$$

This definition and a fuller explanation of the DFT may be found in [Aho 74].

Proof strategy, results.

The strategy of this paper is to focus on the cost of communications in a parallel system. Little consideration is given to the silicon area or time taken to perform arithmetics on "local" data. Instead, the area-time analysis is performed on the area of the connecting wires and the time taken by the on-chip communication of intermediate results. This emphasis will be justified in Sections 4 and 5, when the area and time bounds are derived.

Each possible chip design will be analyzed in terms of the pattern of interconnections it provides between arithmetic units. A well-connected pattern occupies a lot of area, but a poorly-connected network takes a long time to compute a DFT.

One important factor ignored in this paper is the difficulty of performing off-chip communication. If the pinout and gating speed of the chip is fixed, there is clearly a maximum rate at which chip I/O can occur. The time taken to perform an N -point DFT on data initially off the chip is thus $\Omega(N)$, a trivial complexity result. For this reason, the computational problem studied here is the DFT of on-chip data, under the assumption that the timing constants are such that off-chip communication is not critical. An analysis of the achievable performance of multi-chip systems for the computation of the DFT remains as an interesting open problem.

The complexion of the area-time tradeoff in the computation of the DFT may be expressed in two ways. Following [Mead 78], a minimum value may be found for some particular cost function, such as the product of area with time. Alternatively, one may seek a function of area and time that describes the performance of many "good" designs. The result of this paper is expressed in both of these ways.

For cost functions of the form AT^x with $0 \leq x \leq 2$, any chip that performs an N point DFT costs at least $\Omega(N^{1+x/2})$. This minimum is nearly achieved on chips whose arithmetic units are connected in a mesh-type pattern.

The relation $AT^2 > N^2/16$ bounds the performance of any chip of area A that computes an N point DFT in time T . At least two designs come close to this limit: those with either a perfect shuffle or a mesh-type interconnection pattern.

Other approaches.

Area and time considerations have been studied previously. This paper's model is based on the assumptions of [Mead 78], who found an optimal value for the area-time product in VLSI memory chips.

Two studies have been made of the area requirements of interconnection patterns. A random graph on N nodes was found to require $\Theta(N^2)$ area, using a model suggested by the wiring of a printed circuit board [Sutherland 73]. The problem of embedding bipartite graphs in the plane was explored by [Cutler 78].

The theory of cellular automata [von Neumann 66] can be used to elucidate some aspects of the area-time tradeoff. However, cellular automata have only nearest-neighbor connections, making them an inconvenient vehicle for the study of the effect of different interconnection patterns on area and time.

The existence of a space-time tradeoff in a computation of the DFT has been demonstrated [Savage 77]. Unfortunately, it is unclear what connection can be made between the space needed to store intermediate results and the silicon area needed to implement a slower DFT.

Outline.

Section 2 details the VLSI model of computation. In Section 3, a graph-theoretic quantity is defined that will be used to derive lower bounds on area (in Section 4) and time (in Section 5) for chips that perform DFTs. Section 6 concludes with the main result, that $AT^2 > N^2/16$ for any chip with area A that computes an N point DFT in time T .

2. The VLSI Model

The useful area of any VLSI layout can be assigned to one of two functions, either processing or communication. In a loose sense, "processing silicon" combines or stores information while "communicating silicon" transmits it unchanged.

PEs, Wires.

The loci where processing occurs are called Processing Elements, or PEs. The loci of

communication are called interconnections, or wires.

Words.

The basic chunk of information considered in this paper is a word. Words are in one-to-one correspondence with elements of the finite commutative ring over which the DFT is defined. To avoid unedifying detail, the word length (in bits) is treated as a constant in this paper.

Wires, units of length and time.

A wire has unit width and transmits a word from one end to the other in unit time. If the transmission is performed bit-serially, the unit of time is proportional to the word length in bits. If the transmission is word-parallel, the unit of length is proportional to word length.

PEs.

A PE contains at most one word of storage. If larger PEs are envisioned, they must be decomposed into word-sized PEs with connecting wires.

A PE may use words from any number of connecting wires to update its own word in any way, but it may take only a constant amount of time to do so. The functions performed by a PE are thus in $R^k \times R$, where R is the ring used to define the DFT.

A PE may output words onto any number of connecting wires, but may only output its own word or any of the words it received in the last time unit. There is thus no bandwidth limitation on PEs: they may act as many-to-many switches.

There are constants a and t such that no PE occupies more than a units of area nor takes more than t units of time to perform an update on its word.

Nexi.

Wires deliver words to and from a nexus associated with each PE. There is exactly one PE per nexus. Communication between a nexus and its PE is free, costing no area or time.

Each nexus is square in aspect, with side d if d wires connect to it. This ensures that there is more than enough edge length on the nexus to accomodate all connecting wires.

The square shape does entail a large area charge for a nexus of large degree, but in this case its associated PE could be very powerful. It would be permissible, for example, for a PE with degree N to act as a "big switch", permuting N words at a time. Charging $\Theta(N^2)$ area allows enough room for a cross-point; fancier switches with greater delay but less area may be built from small crosspoints. (If a PE with degree N is not a "big switch", it should be decomposed into smaller PEs with lesser degrees. For example, a fan-out of N can be achieved with a tree of $\Theta(N)$ constant degree PEs for a total of $\Theta(N)$ area and $\Theta(\log N)$ delay.)

Input, Output PEs.

An input PE initially contains one of the N elements of the vector that is to undergo Fourier transformation. There are N input PEs.

An output PE will eventually contain one of the N elements of the result of the DFT. The N output PEs are not necessarily distinct from the input PEs.

Wire layout.

Wires are laid out on a grid with unit spacing. Restricting wires to run along grid lines assures that unit width is available for each line if the grid is physically realized with two layers of silicon. One layer is devoted to the "x" direction, one to the "y".

Wires may bend at grid corners. This corresponds to a connection between the two layers of silicon.

At grid corners, wires may cross at right angles with no effect on each other's signals or timing. This corresponds to insulating the two layers of silicon from each other.

Wires may not connect or fan-out at grid corners! A PE is required at the confluence of wires, to avoid non-constant fan-out without area or time cost.

Justification of the model.

The VLSI model of computation is justified if its area and time charges are appropriate, and if it allows for all possible designs.

Area charges can be simply stated: wires have unit area per unit length, and next

have area equal to the square of their degree. The area of PEs is largely ignored, leading to weaker but valid lower bound results. Nexus area is the only arguable charge, and it is unimportant in the usual case of constant fan-out at each PE. In this case, nexus area is only a constant multiple of wire area.

Time charges are even simpler than area charges. Wires transmit one word per unit time. This is consistent with the common VLSI design strategy of matching drivers to their wires. (On-chip propagation speed is limited by wire capacitance, not the speed of light.) PE delays are ignored, which of course cannot invalidate the lower bound results of this paper.

The VLSI model of computation applies to all possible designs, according to the following correspondence scheme. Any chip that performs a DFT must have at least N words of storage for the input vector. The elements that store these values are the N input PEs. Similarly, N output PEs may be identified, even though they may not be distinct from the input PEs. Next, one may identify the wires on the chip by noting the silicon used to carry information from input PEs to other registers on the chip. Finally, the nexi are the switching or gating elements that connect wires to storage elements (PEs) or to other wires.

There is one critical assumption built into the VLSI model, that the information about the N input words is initially localized. That is, each word is stored in a compact region (a PE) of the chip. This assumption is necessary to ensure that the DFT involves some computation, for otherwise one would consider the output words a legitimate initial encoding of the input vector. A similar argument can be made for requiring each of the output words to be stored in its own PE. Localization of the input and output PEs ensures that their nexi are also localized, so that there is indeed exactly one nexus for each PE.

The choice of the word as the basic unit of information is also defensible. Recall that wires of unit width transmit one word in unit time. Wires (and PEs) of smaller capacity are clearly conceivable, and should have fractional width or fractional delay to be true to VLSI implementation costs (for bit-parallel or bit-serial transmission, respectively). The introduction of such fractional capacities would only obscure the results of this paper, not invalidate them. None of the proofs depend upon the integral nature of the degree of a nexus or of the information capacity of any wire.

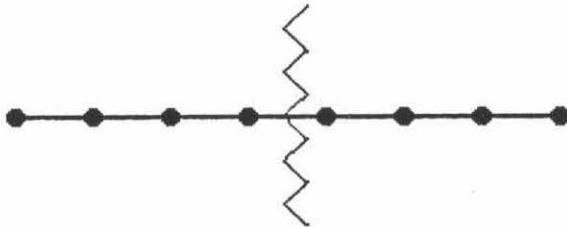
The description of the VLSI model of computation is now complete. It will be seen in the sequel that it is the pattern of interconnections, not the "programming" of PEs, that limits the speed or magnifies the area of a chip.

Interconnection patterns will be analyzed with the aid of the graph-theoretic definitions developed in the next section.

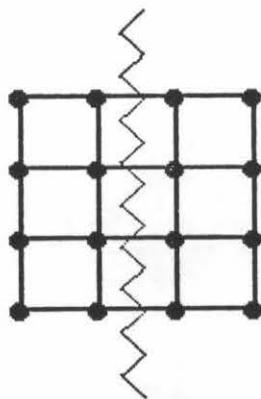
3. Minimal Bisection Width

The minimal bisection width of a graph is, informally, the number of cuts needed to slice it in half. In other words, it is the smallest number of edges whose removal disconnects one half of the vertices from the other.

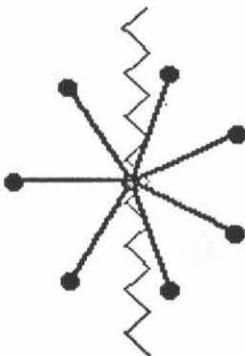
For example, the minimal bisection width of a linear graph of N nodes is 1:



The minimal bisection width of a mesh of N nodes is $N^{1/2}$:



The minimal bisection width of a star of N nodes is $N/2$:



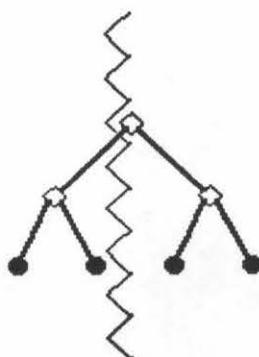
The minimal bisection width of a graph with an odd number of nodes is defined by relaxing the bisection requirement slightly. A bisection of a graph of $2N+1$ nodes splits it into two disconnected subgraphs of N and $N+1$ nodes.

Formally, the minimal bisection width of a graph $G = (V, E)$ is w iff w is the minimum integer satisfying the property that there is a bisection $V = H_1 \cup H_2$ with $H_1 \cap H_2 = \emptyset$, $|H_1| \leq |H_2| \leq |H_1| + 1$, and $|\{(u, v) \in E : u \in H_1, v \in H_2\}| = w$.

Minimal bisection width of a subgraph.

The minimal bisection width of a subgraph is defined in the following fashion for each subset $U \subseteq V$ of the vertices of a graph $G = (V, E)$. The minimal bisection width of U in G is the smallest number of edges in E whose removal disconnects $H_1 \subseteq U$ from $H_2 \subseteq U$, where $|H_1| \leq |H_2| \leq |H_1| + 1$, $H_1 \cup H_2 = U$, and $H_1 \cap H_2 = \emptyset$.

The minimal bisection width of the leaves in a binary tree is 1:



In general, it is difficult to compute minimal bisection widths: the problem is

NP-complete, in fact [Garey 74]. Fortunately, it is enough to know that every graph has a set of edges that realizes its minimal bisection width.

The following sections will derive bounds on area and time for any VLSI design. A graph will be associated with each design, defining a minimal bisection width, w . Lower bounds of $w^2/4$ and $N/(2w)$ will be found on area and time respectively. Thus $AT^2 > N^2/16$.

4. Area

The total area occupied by a VLSI design is the sum of the areas of its PEs, wires, and nexi. A lower bound on wire and nexus area is derived in this section. Inclusion of PE area can at most affect the result by a constant factor, since there is one nexus per PE, and PEs have area bounded by a constant.

Associate with each VLSI design the following graph, G . Each nexus is a vertex. Each wire connecting two nexi is an edge between corresponding vertices. Denote by I the subset of vertices that are nexi of the N "input PEs".

Let w be the minimal bisection width of I in G .

Theorem 1: The area occupied by the wires and nexi of a VLSI design that corresponds to a graph of width w is greater than $w^2/4$.

Proof.

Orient the VLSI design in a Cartesian space. Consider lines of the form $x=a$. Each will separate the nexi of the N input PEs into three subsets: those to the left, split by, or to the right of the line. Denote these sets by L, S, and R. Find a value of a for which $|L| + |S| \geq N/2$ and $|R| + |S| \geq N/2$ (such an a exists by monotonicity). If $|S| = 0$, the line $x=a$ would define a cutset of I in G , hence it would cut at least w edges. Otherwise, split S into two subsets, S_1 and S_2 , such that $|L| + |S_1| = |R| + |S_2| = N/2$. Remembering that a nexus of degree d occupies a square of side d , each nexus in S_1 may be considered to lie wholly to the left of the line $x=a$ (the wires connecting to the nexus on the right side of $x=a$ may be "continued" right up to the $x=a$ border). So the bisecting line still cuts at least w wires.

The line $x=a$ is said to account for the w square units of area of wire and nexus that lie within $1/2$ unit distance of it.

Next, consider "zig-zags" of the form $\{x=a-1 \text{ for } y \geq b, x=a+1 \text{ for } y \leq b, \text{ and } y=b \text{ for } a-1 \leq x \leq a+1\}$. Using the a determined above, vary b to obtain another bisection of the input nexus. Only two wires may cross the horizontal ($y=b$) segment, so that its vertical sections must cross at least $w-2$ wires.

This zig-zag accounts for the $w-2$ square units of wire and nexus that lie within $1/2$ unit of its vertical sections.

In all, $[w/2]$ zig-zags may be drawn, each of the form $\{x=a-k \text{ for } y \geq b, x=a+k \text{ for } y \leq b, \text{ and } y=b \text{ for } a-k \leq x \leq a+k\}$. Each accounts for $w-2k$ area.

The total area of wire and nexus is thus

$$\sum_{0 \leq k \leq [w/2]} w-2k > w^2/4.$$

5. Time

The speed of a VLSI design may be limited either by the time taken by arithmetic operations or by the time taken to get intermediate results to the proper place. It is the latter that generally places the stricter limit on large VLSI designs for the DFT, by the following argument.

As noted in section 2, a PE can take at most t units of time to update its word. The information used to perform this update must have taken at least one unit of time to reach the PE. Thus, within at most a factor of t , it is the routing of intermediate results rather than the arithmetics performed on them, that limits the timing of VLSI designs.

The following theorem places a lower bound on the time required to compute a DFT. Its proof is based on a consideration of the amount of information that must be transmitted during the course of any computation of a DFT.

Theorem 2: Associate a graph with each VLSI design as in Section 4. At least $N/(2w)$ time is required to compute an N point DFT on a VLSI design that corresponds to a graph of width w .

Proof sketch.

The theorem is a consequence of the following property of the DFT. View the DFT as

a matrix-vector multiplication

$$\tilde{y} = A\tilde{x}$$

Bisect the N elements of \tilde{y} into two equal-sized subsets \tilde{y}_1 and \tilde{y}_2 . Partition the elements of \tilde{x} into any two disjoint subsets \tilde{x}_1 and \tilde{x}_2 . This decomposes the DFT problem into two subproblems of the form

$$\begin{aligned}\tilde{y}_1 &= A_{11}\tilde{x}_1 + A_{12}\tilde{x}_2 \\ \tilde{y}_2 &= A_{21}\tilde{x}_1 + A_{12}\tilde{x}_2\end{aligned}$$

It is possible to show that

$$\text{Rank}(A_{12}) + \text{Rank}(A_{21}) \geq N/2 \quad (1)$$

This follows (non-trivially) from an argument on the number of zeros of a polynomial of degree $N/2$. If k of the elements of \tilde{x}_1 are among the first $N/2$ elements of \tilde{x} , then $\text{Rank}(A_{12}) \geq k$. In this case, $\text{Rank}(A_{21}) \geq N/2 - k$.

This property of the DFT holds for any bisection of the elements of \tilde{y} . In particular, it holds for the bisection of the output PEs that realizes w , the minimum bisection width.

Choose \tilde{x}_1 to be the set of input PEs that are included with \tilde{y}_1 in the bisection of the output PEs. The computation of \tilde{y}_1 will require k words of information about \tilde{x}_2 , if $\text{Rank}(A_{12}) = k$. A similar argument holds for \tilde{y}_2 , so that $N/2$ words of information must pass over the w wires that separate \tilde{x}_1 from \tilde{x}_2 .

It takes at least $N/(2w)$ time to pass $N/2$ words over w wires, hence the theorem.

6. Conclusion

Theorems 1 and 2 can be immediately combined to give the main result of the paper.

Theorem 3: If a VLSI design with area A computes an N point DFT in time T , then $AT^2 > N^2/16$.

This theorem is nearly tight for at least two cases. Using a mesh-type interconnection pattern, $O(N)$ space and $O(N^{1/2})$ time is sufficient to compute the DFT. Using a perfect shuffle interconnection, $O(N^2/\log^{1/2} N)$ area and $O(\log N)$ time is sufficient [Pease 68, Thompson 80].

The following theorem is of interest if a particular cost function is to be minimized.

Theorem 4: For any VLSI design with area A and time T, and for any non-negative $x \leq 2$, $AT^x = \Omega(N^{1+x}/2)$.

Proof.

The area of any VLSI design with N input PEs must be $\Omega(N)$ since each PE has a nexus of at least unit area. By Theorems 1 and 2,

$$AT^x = \Omega(N + w^2/4) * (N/2w)^x$$

Without loss of generality, let $w = N^{1/2+\epsilon}$. Then

$$AT^x = \Omega(N^{1+x/2+\epsilon x} + N^{1+x/2+\epsilon(2-x)})$$

Since $0 \leq x \leq 2$, the second term increases with ϵ while the first term decreases with ϵ .

Clearly, $\epsilon=0$ achieves the minimum value, hence the theorem. ■

From the proof of Theorem 4, it is clear that the optimal design has $w=\Theta(N^{1/2})$, which corresponds to a mesh-type interconnection pattern.

A similar analysis may be performed for other problems, including matrix multiplication, Gaussian elimination, transitive closure, sorting, and permutation [Thompson 80].

Acknowledgements.

James B. Saxe formulated and proved equation 1. Leo J. Guibas and the XEROX Palo Alto Research Center gave guidance and generous support during the early phases of the research that led to this paper.

Mere acknowledgement can hardly discharge my debts to my thesis advisor, H. T. Kung. No one else could have given better hints at crucial moments, and I despair of ever finding a more selfless researcher.

References

- | | |
|----------|---|
| [Aho 74] | Aho, Alfred V., Hopcroft, John E. and Ullman, Jeffrey D.
<i>The Design and Analysis of Computer Algorithms.</i>
Addison-Wesley, 1974. |
|----------|---|

- [Bonneau 73] Bonneau, Richard J.
A Class of Finite Computation Structures Supporting the Fast Fourier Transform.
Technical Report 31, Massachusetts Institute of Technology Project MAC, March 1973.
- [Cutler 78] Cutler, M. and Shiloach, Y.
Permutation Layout.
Networks 8:253-278, 1978.
- [Garey 74] Garey, M. R., Johnson, D. S., and Stockmeyer, L.
Some Simplified Polynomial Complete Problems
Proc. 6th Annual ACM Symposium on Theory of Computing, SIGACT, pages 47-63, 1974.
- [Mead 78] Mead, Carver and Rem, Martin.
Cost and Performance of VLSI Computing Structures.
Technical Report 1584, California Institute of Technology Computer Science Department, 1978.
- [Pease 68] Pease, Marshall C.
An Adaptation of the Fast Fourier Transform for Parallel Processing.
Journal of the ACM 15(2):252-264, April 1968.
- [Savage 77] Savage, J. E. and Swami, Sowmitri.
Space-Time Tradeoffs in the FFT Algorithm.
Technical Report TRCS-31, Brown institution, August 1977.
- [Sutherland 73] Sutherland, Ivan E. and Oestreicher, Donald.
How Big Should a Printed Circuit Board Be?
IEEETC C-22:537-542, May 1973.
- [Thompson 80] Thompson, C. D.
A Complexity Theory for VLSI.
PhD thesis, Carnegie-Mellon University Computer Science Department, 1980.
- [von Neumann 66] von Neumann, J.
The Theory of Self-Reproducing Automata.
Univ. of Illinois, Urbana, Ill., 1966.

Direct VLSI Implementation of Combinatorial Algorithms

L.J. Guibas, H.T. Kung, and C.D. Thompson

*Carnegie-Mellon University
Pittsburgh, Pennsylvania*

Abstract

We present new algorithms for dynamic programming and transitive closure which are appropriate for very large-scale integration implementation.

0. THE VLSI MODEL OF COMPUTATION

The purpose of this paper is to give two examples of algorithmic design suitable for direct implementation in VLSI (very large-scale integration). We show new algorithms for two important combinatorial problems, *dynamic programming* and *transitive closure*. In our design we attempt to meet the challenge offered by the new VLSI technology by taking account of its true costs and capabilities. The algorithms of this paper meet the goals of modularity and ease of layout, simplicity of communication and control, and extensibility. These goals are of paramount importance in all VLSI designs.

Modularity and ease of layout: The design of significant system components using large-scale integration is notoriously expensive. Two major factors of the design cost are the difficulty of designing each chip, and the number of different chips that must be designed. A modular design decreases both cost factors, as well as facilitating chip and system layout. An ideal structure for VLSI has a large number of identical modules organized in a simple, regular fashion. The resulting ease of layout dramatically reduces design costs, accounting for the successful use of VLSI in memory and PLA chips.

We have taken a somewhat extreme approach to system modularity by proposing a single simple module, called a *cell*. Cells are laid out in a planar array, with connections only to nearest neighbors. The layout of a chip is thus trivial, as is the layout of chips on a board. Our cells combine memory and processing in a finer grain than has been customary. A device built from such cells can perform a substantial computational task, even though it has a topology much more like that of a passive memory than a von Neumann microprocessor.

We restricted our attention to cells that could be implemented with at most a few thousand active elements (gates, transistors). Many modules may thus be laid out on a single VLSI chip, giving structure to the chip design problem. Furthermore, our algorithms can make efficient use of ten to ten thousand or more cells, so that many identical chips can be used in one installation.

Communication and control: For a VLSI design to be truly practical, it must not sidestep any communication or control issues. A good design minimizes both the complexity of each module as well as the number of connecting wires between modules. The latter consideration becomes more important as VLSI technology improves. An increase in the number of active elements on a chip is of little benefit when the chip's I/O bandwidth is limited by its pinout.

We considered designs with eight connections to each cell: power, ground, clock, reset, and four data lines to neighboring cells in the planar array (left, right, up, down). We then attempted to minimize the solution time for a dynamic programming (or transitive closure) problem, assuming a data rate of one bit per clock cycle.

A VLSI chip has enough pins to implement many of our cells, if the cells form a square region of the planar array. For example, 9 cells in a 3×3 array need only 16 external connections: 12 data lines for the cells on the periphery, and 4 common wires for power, ground, clock and reset.

Similarly, a 5×5 matrix will fit on a 24 pin chip, an 8×8 on a 36 pin chip, and a 9×9 on a 40 pin chip. This is the limit of present technology, though 100 pin chips are conceivable in the future. With a few thousand active elements per cell, our designs are of appropriate complexity for VLSI.

Another consideration is the balancing of on-chip processing with I/O. Input and output are fundamentally slower than communication on the chip. This suggests that the construction of custom devices will only be economical for the implementation of "super-linear algorithms", where on-chip processing is of sufficient complexity to balance the time required to read in or write out the data. In other words, our aim is to take an algorithm of more than linear complexity in the classical serial model (say $O(n^3)$) and speed it up by the use of parallelism and pipelining, so that the resulting device can process the data at roughly the same rate that data can be input or output.

Extensibility: This paper deals with the design of special purpose hardware to solve two specific classes of problems. The utility of such designs is limited by their specificity. We seek extensibility in two ways, through size and problem independence.

The hardware should be able to solve arbitrarily sized instances of the problem for which it is intended. Here we suppose that our designs are used to build special purpose devices controlled by a more conventional processor. In this light, it is important that the devices can be used efficiently for the solution of problems that exceed their (one-pass) capacity. This issue will be further explored in Section 3, under the rubric of *decomposability*.

Problem independence is even more important: special purpose hardware should solve as many different problems as possible. Modules could conceivably be microprogrammed as logic density increases. This paper indicates the utility of the mesh-style interconnection pattern for modules, as well as demonstrating two (perhaps generally useful) patterns for data flow in such systems.

Systolic algorithms: There is a newly coined word for our style of algorithm design for VLSI: "systolic", in the sense of [KL2]. The term is meant to denote arrays of identical cells that circulate data in a regular fashion. Kung and Leiserson's cells [KL, KL2] perform but one simple operation; we have relaxed this restriction somewhat so that a small amount of control information circulates with the data.

Data movement is synchronous and bit-serial, to reduce pinout requirements. It is a difficult but hardly insurmountable problem to design chips with ten to one hundred identical modules synchronized with a single clock line.

Time is measured in terms of data transmission. One "word" may be sent along a wire in one unit of time. This convention avoids extraneous detail in the discussion of our algorithms: the choice of word length is left to the implementor. It unfortunately obscures one important issue, namely, that one bit of control information must be sent with each word of data in the transitive closure and dynamic programming algorithms.

Traditional models and goals: Algorithmic design is dominated by the traditional goals that arise naturally from classical machine architectures and technologies. A good serial algorithm minimizes

operation counts, while a good parallel algorithm maximizes concurrent processing. Both viewpoints are somewhat inappropriate to the evaluation of VLSI designs. The theory of cellular automata is more helpful.

In a typical serial model, $o(n^3)$ boolean operations are sufficient to compute a transitive closure [AHU, Chap. 6]. However, the recursive algorithm employed does not seem suitable for direct VLSI implementation, since too much information is passed across recursive calls.

A parallel algorithm has optimal speedup if it cuts computation time by a factor proportional to the number of processors used. But computation time is measured in most parallel models by counting elementary operations, with little consideration of the time necessary to transmit intermediate results.

The theory of cellular automata [vN] does offer some insight into VLSI design. Its generality obscures some important issues, for example, the cost of building the cells (especially if there is more than one type), and the amount of information received by a cell from its neighbors in one unit of time. It lacks the notion of the I/O bottleneck between chips due to pinout restrictions. And it does not address the problem of decomposability.

Other work: Models of computation similar to ours have been previously considered in the literature. This paper was inspired by Kung and Leiserson's [KL] solutions to several matrix problems, and by the vision of VLSI expressed in Mead and Conway's book [MC]. Levitt and Kautz [LK] explored the hardware implementation of Warshall's [Ws] algorithm for transitive closure. However, their designs are not readily decomposable, and they use many more connections per cell.

Organization of the paper: Algorithms for dynamic programming and transitive closure are developed separately in sections 1 and 2. Section 3 concludes the body of the paper with a discussion of decomposability and further topics.

1. DYNAMIC PROGRAMMING

Many problems in computer science can be solved by the use of dynamic programming techniques. These include shortest path, optimal parenthesization, partition, and matching problems, and many others. For a fuller discussion of this spectrum see the review article by K. Brown [Br] and the references mentioned therein. In this paper we will confine our attention to optimal parenthesization problems. This will allow us to explain the ideas without excess generality, while at the same time covering a vast range of significant problems, including the construction of optimal search trees, file merging, context free language recognition, computation of order statistics, and various string matching problems.

The optimal parenthesization problems can all be put in the form:

Given a string of n items, find an *optimal* (in a certain sense) parenthesization of the string.

As an example, there are five distinct parenthesizations of the string (1 2 3 4):

$$(((1\ 2)\ 3)\ 4) \quad (1.1)$$

$$(((1\ (2\ 3))\ 4)) \quad (1.2)$$

$$((1\ 2)\ (3\ 4)) \quad (1.3)$$

$$(1\ ((2\ 3)\ 4)) \quad (1.4)$$

$$(1\ (2\ (3\ 4))) \quad (1.5)$$

Note that a parenthesization of n items has $n - 1$ pairs of parentheses. Also, each parenthesis pair encloses two elements, each of which is either an item or another parenthesis pair. These five parenthesizations correspond to the five ways of performing the addition $1+2+3+4$ without rearranging the terms.

The optimality of a parenthesization is defined with respect to a cost for each parenthesis pair; the total cost is some function (usually the sum) of the individual costs. The optimal parenthesization is the one with minimum total cost.

In the previous example, if the cost of a parenthesis pair is defined as the sum of the enclosed items, then the optimal parenthesization is the first one, with total cost 19.

The obvious way of solving an optimal parenthesization problem involves examining all possible parenthesizations of the string, then picking the one with the smallest cost. This algorithm is clearly exponential in n , as the number of distinct parenthesizations is itself exponential. A dynamic programming strategy for this problem is derived from the following observations. If we have an optimal parenthesization of the whole string, then we also have an optimal parenthesization of each of its substrings. (If we could improve a substring, then we could improve the whole). This suggests that we calculate the optimal parenthesizations of successively larger substrings of the original string, starting from the singleton items. Further, we note that the solution for a given substring will arise as a subproblem for several larger problems, and thus it will pay to remember the optimal solution in order to avoid recomputation.

To simplify matters suppose that we are only interested in the cost of the minimum cost parenthesization, not its structure. Let the original string items be numbered by the integers 1 through n from left to right, and let $c(i, j)$ denote the minimum cost of parenthesizing the substring consisting of items i through $j - 1$ (here we assume $1 \leq i < j \leq n + 1$). Then, according to the above discussion, the $c(i, j)$ can be computed using a recurrence of the form

$$c(i, j) = \min_{i < k < j} F_{ikj}(c(i, k), c(k, j)). \quad (1.6)$$

Here, the $c(i, k)$ and $c(k, j)$ are the optimal costs for parenthesizing two substrings. The range of the minimization variable, k , ensures consideration of all pairs of substrings that can be formed from items i through $j - 1$. The function F_{ikj} computes the total cost of parenthesizing the items

i through $j - 1$, and it is normally of the form

$$F_{ikj}(c(i, k), c(k, j)) = c(i, k) + c(k, j) + f(i, k, j). \quad (1.7)$$

The total cost of parenthesizing items i through $j - 1$ is in this case the sum of the optimal parenthesization of a pair of substrings, plus some additional cost for the outermost pair of parentheses.

In our toy example delineated in the text above,

$$c(1, 2) = c(2, 3) = c(3, 4) = c(4, 5) = 0, \quad (1.8)$$

and

$$F_{ikj}(c(i, k), c(k, j)) = c(i, k) + c(k, j) + \sum_{i \leq h < j} h. \quad (1.9)$$

The desired value is $c(1, 5)$, the cost of parenthesizing items 1 through 4. The dynamic programming approach to evaluating $c(1, 5)$ is to solve all "subproblems" first. Thus,

$$c(1, 3) = c(1, 2) + c(2, 3) + (1 + 2) = 3, \quad (1.10)$$

$$c(2, 4) = 5, \quad (1.11)$$

$$c(3, 5) = 7. \quad (1.12)$$

With these values known, the following may be computed:

$$c(1, 4) = \min(3 + 6, 5 + 6) = 9, \quad (1.13)$$

$$c(2, 5) = 14. \quad (1.14)$$

Finally,

$$c(1, 5) = \min(9 + 10, 14 + 10) = 19, \quad (1.15)$$

as asserted previously. The optimal parenthesization is that of (1.1), as identified by the optimal values of k found in applications of (1.6).

In general, one may compute the $c(i, j)$ in order of increasing value of $j - i$, ending with the computation of $c(1, n + 1)$.

Note that since there are $\Theta(n^2)$ substrings of the original, and for each substring we are minimizing over $\Theta(n)$ values of k on the average, the computation takes a total of $\Theta(n^3)$ steps. Thus dynamic programming has given us a low order polynomial algorithm for an apparently exponential problem.

We visualize a more general computation by using the triangle depicted below for the case $n = 6$.

[Figure 1.1]

Denote by (ij) the solution to the parenthesization problem for the substring from i to $j - 1$. We start out with $(12), (23), \dots, (67)$, the singleton substrings, whose solution we assume is given.

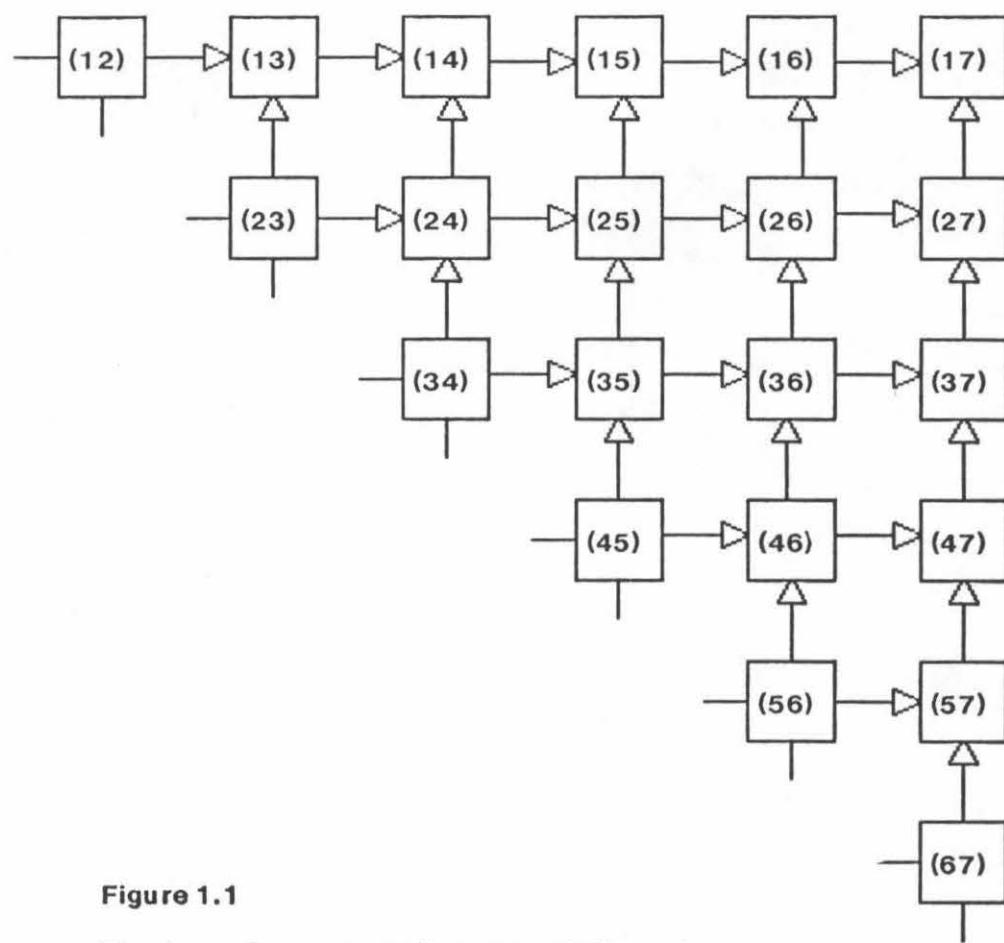


Figure 1.1

The dynamic programming computation.

Using these we can then compute the solutions for substrings of length 2, namely (13), (24), ..., (57). Next we can get (14), (25), ..., (47), then (15), (26), (37), then (16), (27), and finally the desired result (17).

The above serial algorithm is amenable to certain obvious parallelism. If all (ij) with $j - i < s$ are available, then the (ij) with $j - i = s$ can be computed in parallel. Thus if we had $\Theta(n)$ cells, and ignored the cost of data movement, we could finish the computation in $\Theta(n^2)$ steps. This decomposition is, of course, much closer to classical parallel models than to the VLSI model we are advocating. Note that each cell is working in isolation on a complete sub-problem. Previous results must be made available to several cells and thus, unless we design the data movement carefully, contention problems may arise. By contrast, we are seeking an algorithm with simple and regular data flow which offers maximal pipelining. For us cells are inexpensive, as long as they are of a simple and uniform kind. We are happy to provide $\Theta(n^2)$ cells, if we can then solve the problem in $\Theta(n)$ steps.

We now drop the toy example for a more realistic problem, for the remainder of this section. The problem we wish to tackle is that of the construction of an optimum binary search tree [Kn, p.433], for which the above recurrence becomes

$$c(i, j) = w(i, j) + \min_{i < k < j} (c(i, k) + c(k, j)), \quad (1.16)$$

where $w(i, j)$ is the sum of the probabilities that each of the items i through j will be accessed. We will try to solve this problem on a network of cells suggested by Figure 1.1, that is, a triangle of $n(n + 1)/2$ cells connected along a rectangular mesh. Cell (ij) will compute one number, the value of $c(i, j)$. Then, it will send its result to the cells that need it to compute their own value.

Note that this structure satisfies many of the *a priori* requirements for efficient VLSI implementation. We have a simple and regular interconnection pattern that corresponds well to the geometrical layout. Furthermore only the cells along the diagonal, and the cell at the upper right hand corner need communicate data to or from the outside world, thus guaranteeing a reasonable pin count. However, much remains to be worked out with respect to data flow. According to our recurrence, for example, cell (17) will need to combine the result of (16) (one of its neighbors) with the result of (67) (a cell far away). The art in the design of this algorithm lies in arranging for the right data to arrive at the right time at each cell, without overloading the communication paths.

As noted in the introduction, time in our systems is defined by data transfers. One unit of time is sufficient for the communication of the value of a $c(i, j)$ between neighboring cells. (Eventually, it will be seen that *two* $c(i, j)$ values and one bit of control information must pass in unit time over the single wire connecting neighboring cells. If additional pinout is available, this bit-serial transmission may be parallelized in the normal fashion.)

We now explain our algorithm. For simplicity of exposition we assume that cell (ij) has been preloaded with $w(i, j)$. (In fact this loading operation can be merged with the computation described

below). Let us say that cell (ij) is at distance $j - i$ away from the boundary (e.g., the diagonal (11), ..., (77)). An informal description of the algorithm can then be given as follows:

If a cell is at distance t away from the boundary, then its result is ready at time $2t$. At that moment the cell starts transmitting its result upwards and to the right. The result travels along both directions by moving by one cell per time unit, for t additional time units. From that moment until eternity the result slows its movement by a factor of 2. (That is, it now moves to the next cell every two time units).

Before we descend into the details of how to implement this data flow pattern, let us see that it causes all the right data to arrive at a cell at the right time. A proof of this can be given formally, but is best illustrated by an example: cell (17). The first pair that this cell can hope to combine is (14) and (47) (every other pair has some member that will be generated later than these two). Both (14) and (47) will be ready at time $2 \cdot 3 = 6$, as they are a distance of 3 away from the boundary. They will travel at full speed for 3 more time units along the paths towards (17), arriving there at time 9. Now we claim the at time 10 cell (17) will be able to combine the two additional pairs (15) with (57), and (13) with (37). We may check just the first one, as the other is clearly symmetric. The result of (15) is available (by our assumption) at time 8, and thus will arrive at (17) at time 10. But (57) is more interesting. It will be ready at time 4, will travel towards (17) at full speed for 2 more time units, arriving at (37) at time 6. But now it will slow down by a factor of 2, and thus it will need 4 more time units to get to (17), arriving there at time 10! Similarly, at time 11, our cell will be able to combine (16) with (67), and (12) with (27). This is all for the good, because thus at time 12 cell (17) is ready to start transmitting its result, exactly as our scheme would require, since it is a cell at distance 6 away from the boundary. For a network of size n , $2n$ time units will be required before the final result is available.

After this overview of the algorithm, we must now examine the implementation at greater depth and check that the available communication paths are adequate to carry the data flow required.

It is simplest to divide the capacity of the wire connecting adjacent cells into three channels. We call these the *fast belt*, the *slow belt*, and the *control line*. The first two channels carry one $c(i, j)$ value per unit time, whereas the *control line* transmits one bit per unit time. (We defer discussion of the control line until the action of the cells has been completely described.)

The cells make use of their communication channels in the following manner. Each cell has five registers: the *accumulator* where the current minimum is maintained, the *horizontal fast* and *horizontal slow* registers, and similarly the *vertical fast* and *vertical slow* registers. On its horizontal *fast belt* connection, a cell normally receives the contents of its left neighbor's horizontal fast register (storing it into its own horizontal fast register), while passing the old contents of that register to its right neighbor. The horizontal slow belts behave in exactly the same way, except that the horizontal slow registers consist of two stages. The data coming in enters the first stage, moves to the next stage at the next time unit, and finally exits the cell at the following time unit. The nomenclature should

now be clear: data in the *fast belt* moves one cell to the right every time unit, while data in the *slow belt* only moves by half to the right every time unit. Completely analogous comments apply to the movement of data upwards in columns of vertical registers.

The operation of a cell is then the following. During every unit of time a cell partakes in the belt motion and also updates its accumulator. The new value is the minimum of the old accumulator contents, the sum of the new contents of its horizontal fast and vertical slow registers, and the sum of its horizontal slow and vertical fast registers. In addition, if this cell is at distance t away from the boundary, then at time $2t$ it will copy the contents of its accumulator into its fast horizontal and vertical registers. And finally, if it is at an even distance $t = 2s$ from the boundary, then at time $3t/2$ it will load the first stage of its horizontal (and vertical) slow register from the horizontal (resp. vertical) fast belt, ignoring its slow belts entirely.

We see that our algorithm uses only a bounded number of registers per cell, thus meeting another of the desired attributes of a solution. In order to prove that this works we must show that for every belt no data gets overwritten which still needs to be used. Figure 1.2 below illustrates the contents of the fast and slow horizontal belts for the first row of cells in the example discussed earlier.

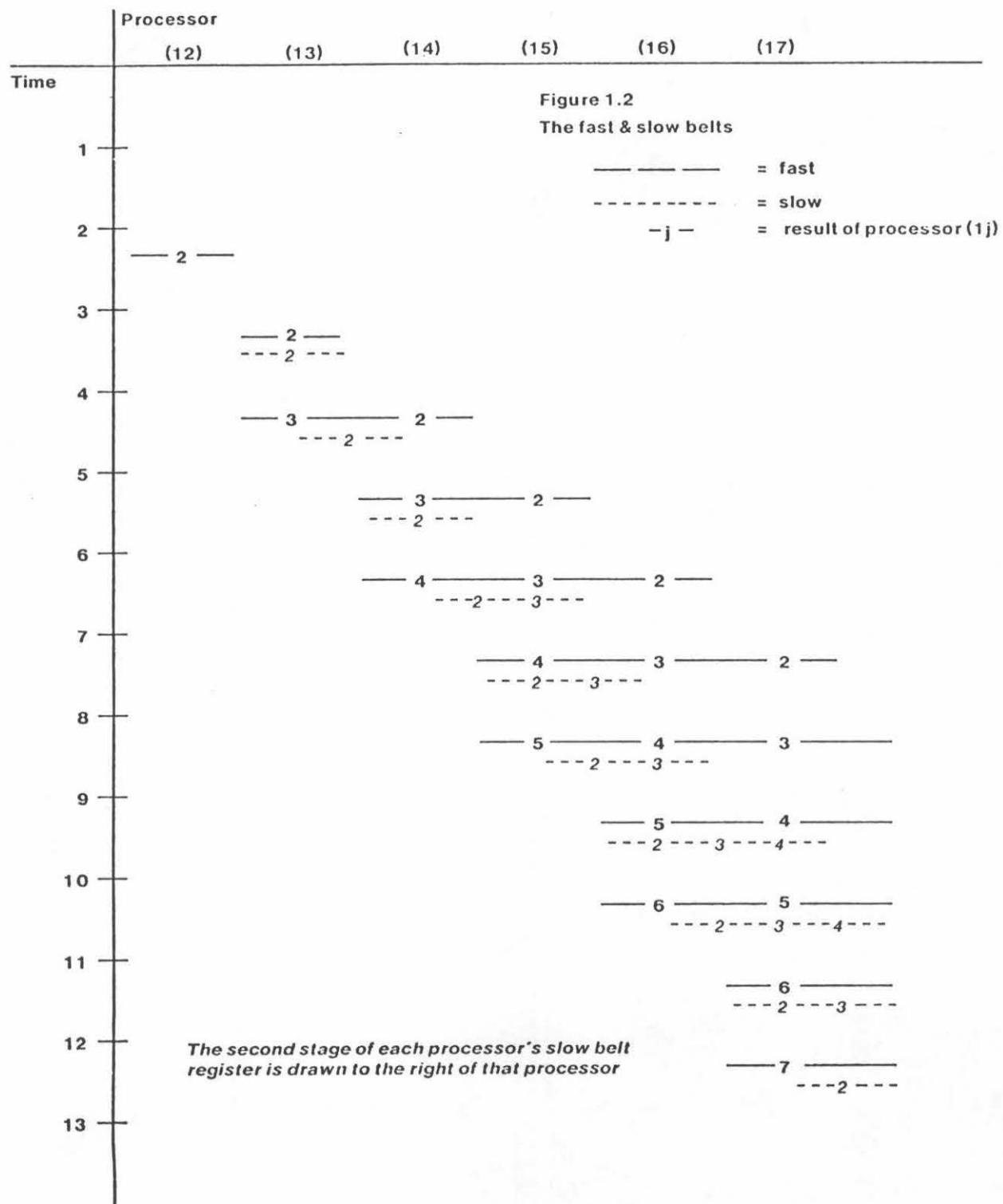
[Figure 1.2]

Notice that when a new value is stored on a belt, it is never on top of a previously written value. In addition the fast belt, in a sense, "reverses" in space the results the cells write on it. In other words, the results occur on this belt in the opposite direction from the direction that the corresponding cells are laid out in space. In contrast, the slow belt maintains the ordering of the results the same as that of the cells. This sheds some light as to why the two speed scheme succeeds in allowing a cell to combine results generated "close by" with results generated "far away".

At last we need to address the timing issue: does a cell need to know its distance from the boundary and count accordingly? The answer is no, the signals on the *control lines* are sufficient to determine the action of each cell in a uniform fashion. These lines have a capacity of one bit per time unit. During each time unit, a cell receives one control bit from the left and one from below, and transmits one bit to the right and one bit to its upward-adjacent neighbor.

The control signals "flow" through the system, much as the data does, although at a different rate. The accumulator to fast belt transfer that occurs in cell (ij) at time $j - i$ is controlled by a rightward moving signal that moves at a rate of one cell every two time units. The fast to slow belt transfer that occurs at time $3(j - i)/2$ is controlled by an upward moving signal that moves at a rate of two cells every three time units.

We end this section by summarizing again the important attributes of our cell. First of all, there is only one kind. It has a small number of registers and small number of data and control paths connecting it to its immediate geometric neighbors. And finally, both the architecture of the cell and the algorithm it executes are independent of the network size.



2. TRANSITIVE CLOSURE

The transitive closure problem is the following. Consider an $n \times n$ matrix A of 0's and 1's. We interpret A as defining a directed graph on the n vertices 1, 2, ..., n . The (ij) entry of A is 1, if and only if there is a directed arc in the graph from vertex i to vertex j . The transitive closure of A , to be denoted by A^* , is also an $n \times n$ 0-1 matrix, whose (ij) entry is a 1 iff there is a directed path from vertex i to vertex j in the graph. Formally speaking, there is a directed path from i to j iff 1) there is a directed arc from i to j , or 2) there is a vertex k for which there are directed paths from i to k and from k to j , or 3) if $i = j$.

The transitive closure problem arises in many contexts in computer science. In implementing process synchronization, when a resource becomes available, we must trace down chains of processes each suspended on a resource held by another in order to discover which may run next. In updating a computer display containing objects partially obscuring other objects, we again must compute the closure of the "in front of" relation. In the data-flow analysis of computer programs we often need the closure of the "call" relation. Tree or graph traversal (such as garbage collection) can also be viewed as transitive closure problems. Dijkstra [D] has argued that transitive closure should be one of the fundamental building blocks in any programming system. He pointed out several other problems in the solution of which the computation of a transitive closure naturally arises. Furthermore, as became clear in the last section, what really defines our algorithms is data movement and not data operations. This implies that any network we construct for transitive closure is likely to be also applicable to any other problem with the same data flow. A large class of such problems, called shortest path problems, is discussed in [AHU, Sect. 5.6-5.9].

There is a well-known serial algorithm for the transitive closure problem due to Warshall [Ws]. Subsequently Warren [Wn] published an interesting "row-oriented" algorithm. Both of these algorithms compute the transitive closure of an $n \times n$ matrix in time $\Theta(n^3)$. It is also well known that the serial complexity of transitive closure and matrix multiplication are the same. Thus, at the expense of much more complicated code, the asymptotic complexity of the problem can be further reduced, using the techniques of Strassen or Pan. In this section we will seek a simple $O(n)$ algorithm implementable on a square mesh of n^2 cells. The transitive closure problem has in fact been previously considered by cellular automata theorists and two solutions are known to the authors, one by Christopher [Ch], and one by van Scoy [vS]. Both of these operate in $O(n)$ time. However, they lack certain essential ingredients of an algorithm appropriate for VLSI implementation. First, the complexity of the program executed by a cell is high. In both papers the code expressed in pseudo-ALGOL is over four pages long. Second, and more important, these algorithms have bi-directional data flow along both the horizontal and vertical connections. As we will see in the next section, this makes it quite difficult to *decompose* the algorithm, that is to implement it when only a $k \times k$ array of cells is available, with k less than n .

A useful device for the correctness proof for many of the above algorithms is the notion of "versions". This is especially appropriate when we imagine that we are updating each entry a_{ij} of A in place.

We introduce versions $1, 2, \dots, n$ for each element a_{ij} , where version 1 is the original a_{ij} , and version $n + 1$ the (ij) entry of the transitive closure. In terms of the graph model, we interpret the t -th version of a_{ij} , written as $a_{ij}^{(t)}$, to denote the existence of a path from vertex i to vertex j which, aside from its starting and ending vertices, only visits vertices in the set $\{1, 2, \dots, t - 1\}$. This interpretation makes clear that the $a_{ij}^{(t)}$ can be computed from the recurrence

$$a_{ij}^{(t+1)} = a_{ij}^{(t)} + a_{it}^{(t)}a_{tj}^{(t)}, \quad (2.1)$$

where we use “+” to denote logical *or* and product to denote logical *and*. The above recurrence indicates a partial order according to which versions of different elements must be computed. Both Warshall's and Warren's algorithms can be viewed as certain natural “topological sorts” of this partial order. The same machinery will be useful for justifying the algorithm suggested below. As a final point, note that the values of $a_{ij}^{(t)}$ are monotonic increasing in t , and thus wherever version t is required in the right-hand side of the above recurrence, it is always safe to use version s , if $s \geq t$.

We now describe our solution. We use an $n \times n$ array of cells with the rectangular mesh connections. End-around (toroidal) connections are useful but not essential. Each cell has an accumulator initialized to 0 (false). We also use some external memory that can hold a copy of A . We visualize two copies of the array A flowing past this processor array, one copy horizontally and the other copy vertically, as suggested in Figure 2.1.

[Figure 2.1]

Note that the horizontal copy is a vertical mirror image of A and that it is tilted backwards by 45 degrees. Analogous comments apply to the vertical copy. The tilting of the copies is used so that element a_{i1} from the horizontal copy and element a_{1j} from the vertical copy arrive at the cell in location (ij) at the same time. The algorithm now proceeds as follows (for simplicity we assume here the existence of the end-around connections):

During each time unit, the horizontal and vertical copies advance by one to the right and down respectively. Each cell *ands* the data coming to it from the left and above and *ors* it into its accumulator. Normally a cell passes the data coming from the left to its right, and that from above, downwards. However, when an element of the horizontal copy passes its *home location* in the cell array, it updates itself to the value of the accumulator at that location. Thus when element (32) of the horizontal copy enters cell (32) on the left, the contents of that cell's accumulator exit on the right. As the horizontal copy starts coming out at the right end of the cell array, it is immediately fed back in at the left using the end-around connections. Entirely analogous comments apply to the vertical copy. After the two copies have cycled thus *three* times through, the accumulators of the cell array contain the transitive closure A^* of A (stored in the standard row-major order). The result can now be unloaded by a fourth pass like the above, or by a separate mechanism.

The correctness of the algorithm can be proven by using the idea of versions discussed earlier. Observe that over each cell, during every time unit, the column index of the element currently there

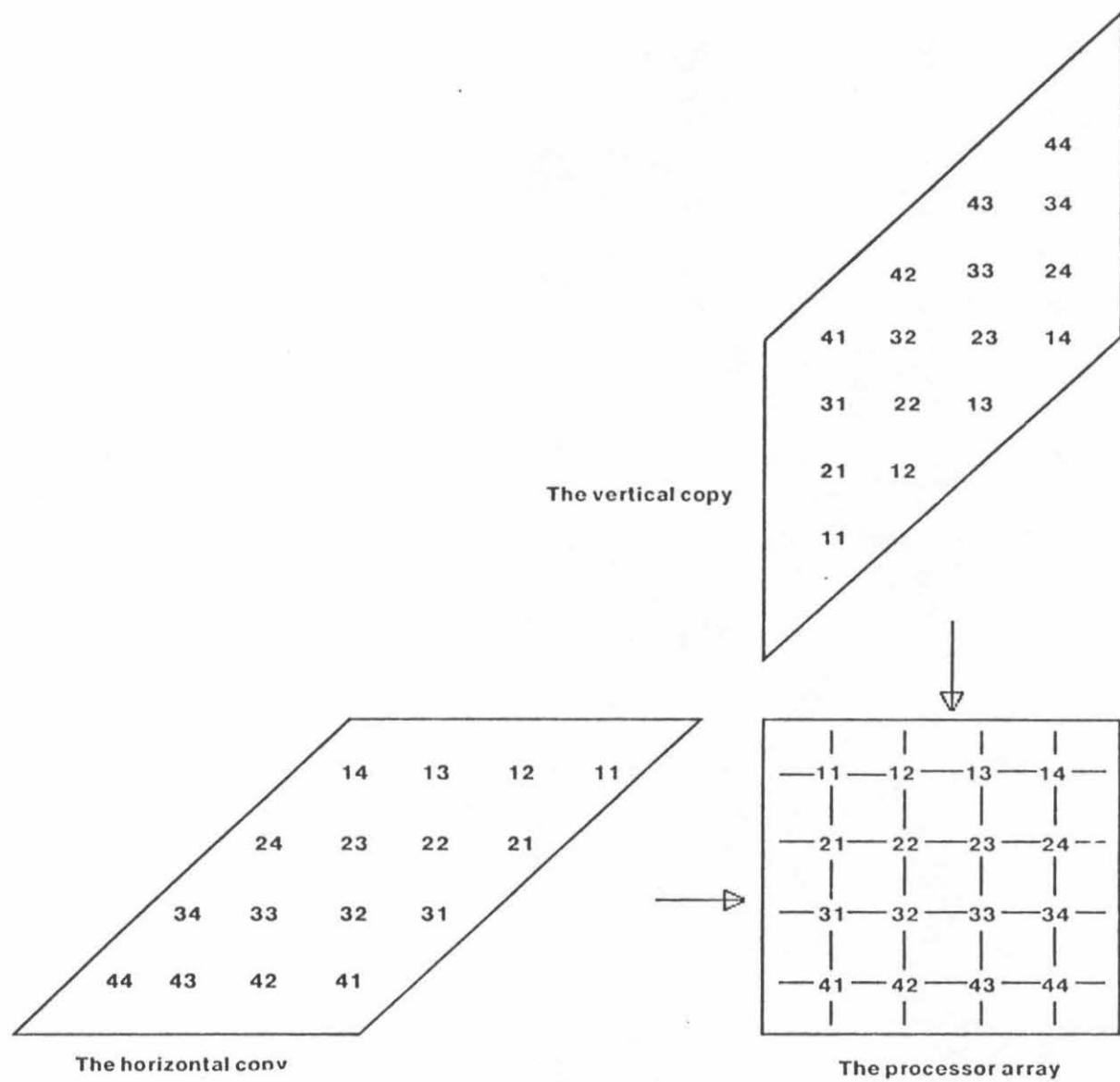


Figure 2.1
Transitive closure

from the horizontal copy equals the row index of the element from the vertical copy. Inductively this implies that the contents of accumulator (ij) are always majorized by transitive closure entry a_{ij}^* (they cannot ever become 1 if the corresponding transitive closure entry is 0). So it is necessary to show only that the accumulator of cell (ij) is brought up to version $n + 1$ of element a_{ij} .

We claim that, after the first pass of the two copies over the cell array, element a_{ij} is brought up to version $p = \min(i, j)$ in both copies. To see this note that equation (2.1) implies that

$$a_{ij}^{(p)} = a_{i1}^{(1)}a_{1j}^{(1)} + a_{i2}^{(2)}a_{2j}^{(2)} + \dots + a_{i(p-1)}^{(p-1)}a_{(p-1)j}^{(p-1)}. \quad (2.2)$$

If we inductively assume that our claim is true for elements in either copy of the form $a_{ij'}$ with $j' < j$, or elements of the form $a_{i'j}$ with $i' < i$, then we can conclude that cell (ij) will perform the inner product in (2.2) above, *before* the (ij) entry of either copy has passed over it. Thus by induction and the monotonicity of the *or* operation our claim is proved. A slight modification of the same argument proves that, after the second pass, element a_{ij} is brought up to version j in the horizontal copy and version i in the vertical one. Finally, if we use (2.2) with $p = n + 1$ then we can conclude that, after the third pass, the accumulator of cell (ij) will contain the ij -th entry of the transitive closure.

As in the case of dynamic programming, it remains to discuss the implementation of timing. How does a cell know when one of its "mates" is passing over it? Once again, this problem can be solved by including one bit of control information with each datum transmitted in the array of cells.

Let's confine our attention to the horizontal mates, as the situation for vertical mates is entirely dual. Note that the horizontal mate arrives at cell (ij) exactly when the diagonal element (jj) of that column in the vertical copy arrives there also. This suggests an extremely simple timing implementation. We start enabling signals at the top edge of the cell array which propagate downwards. The signals move by one cell during each unit of time. Furthermore, we start the signal at cell $(1j)$ at time $2 * (j - 1)$. It is easy to check that these signals coincide with the diagonal elements of the vertical copy. Thus here also our cells execute code independent of the network size.

The overall time required to complete the computation (including unloading of the cell array) is easily seen to be $5 * (n - 1)$, the same as the best previously known solution for cellular automata [Ch]. However, a direct implementation of that solution in VLSI would be inferior to our algorithm, as the units of time are different: more control information flows between cells during each time unit in Christopher's solution.

3. ALGORITHM DECOMPOSITION AND FURTHER TOPICS

We now take up some additional issues. First is the problem of algorithm decomposition. We explore here this issue in the context of the transitive closure problem. Similar comments apply to dynamic programming. If we are given a $k \times k$ array of cells and want to compute the transitive closure of an $n \times n$ matrix, with $k < n$, how do we do it? For simplicity we suppose that k divides

n . Thus we can evenly divide our $n \times n$ matrix into $k \times k$ blocks. To process a block we will cycle through it a horizontal and a vertical section of the array, using the algorithm of the previous section. From the horizontal copy of the full array we extract the $k \times n$ slice corresponding to the block rows and feed that into the device horizontally. Similarly, from the vertical copy we extract the $n \times k$ vertical slice corresponding to the columns of the block. As the slices flow out of the device, they update the memory in which the corresponding array copies are stored. The $k \times k$ blocks can then be processed in this fashion in any order consistent with both the left-right and the top-down ordering of the blocks (the Young tableau order). Many variations on this basic idea are possible, including interleaving the processing of the blocks with the three passes, regenerating one of the slices on the fly so it need not be stored, and others. Recent work of Mead and Rem [MR] on LSI implementations of arrays so they can be accessed either by row or by column has applications here.

The correctness of the decomposition can be proved by using the "monotonicity of versions" remark in the previous section. The case $k = 1$ gives an interesting serial algorithm, which can be viewed as the next logical step in the sequence whose first two terms are Warshall's and Warren's algorithms (in this order). Note that the computation time is now $O(n^3/k^2)$, and thus we still have optimal speed-up to within a constant factor. Finally we remark that a decomposition such as the above is possible precisely because we have signals flowing only downwards and to the right. This leads to an acyclic dependency graph, since there is an order in which to process the blocks such that each computation depends only on previous computations. If we had bi-directional signals along some dimension, so that there exist two blocks along that dimension each depending on signals from the other, then we could not complete the processing of either block without starting the other. Although it is still possible to run the two blocks as coroutines, the complications of saving state and loading and unloading the device would make such a solution prohibitively expensive.

We now conclude with some more general remarks. There is substantial similarity between the dynamic programming and transitive closure cell. Even stronger is the similarity between the transitive closure cell and that used in Kung and Leiserson's work on matrix algorithms. Both are "inner product step" cells. The possibility of mapping all these algorithms onto one type of module needs further exploration.

From a mathematical point of view, perhaps the most interesting question is to ask for a characterization of the computations which can be carried out in this style within certain performance bounds. If we start from a recurrence describing a serial algorithm for the solution to a problem, is there a theory to help us in designing a network like those described here, which would execute exactly the same computation steps, only in a highly parallel and pipelined fashion? Can we describe what processor topologies can be used for what kind of recurrences? The number of open questions is vast.

4. REFERENCES

- [AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974

- [Br] Brown, K.Q., *Dynamic programming in computer science*, CMU tech. rep., June 1978
- [D] Dijkstra, E.W., *Determinism and recursion versus non-determinism and the transitive closure*, EWD456, October 1974, Burroughs Corp.
- [Ch] Christopher, T.W., *An implementation of Warshall's algorithm for transitive closure on a cellular computer*, rep. no. 36, Inst. for Comp. Res., Univ. of Chicago, February 1973
- [KL] Kung, H.T., and Leiserson, C.E., *Algorithms for VLSI processor arrays*, Symp. on Sparse Matrix Computations, Knoxville, Tn., November 1978
- [KL2] Kung, H.T., and Leiserson, C.E., *Systolic Arrays for VLSI*, Cal Tech Conference on VLSI, Pasadena, Ca., January 1979
- [Kn] Kunth, D.E., *The Art of Computer Programming*, vol. 3, Sorting and Searching, Addison-Wesley, 1973
- [LK] Levitt, K.N., and Kautz, W.H., *Cellular arrays for the solution of graph problems*, Comm. ACM, Vol. 15, No. 9, (1972), pp. 789-801
- [MC] Mead, C.A., and Conway, L.A., *Introduction to VLSI Systems*, textbook in preparation
- [MR] Mead, C.A., and Rem, M., *Cost and performance of VLSI computing structures*, Calif. Inst. of Tech. tech. rep., June 1978
- [vS] van Scy, F.L., *A parallel transitive closure algorithm requiring linear time*, unpublished manuscript, May 1977
- [vN] von Neumann, J., *The theory of self-reproducing automata*, (Burks, A.W., editor), Univ. of Illinois, 1966
- [Wn] Warren, S.W., Jr., *A modification of Warhsall's algorithm for the transitive closure of binary relations*, Comm. ACM, vol. 18, no. 4, April 1975, pp.218-220
- [Ws] Warshall, S., *A theorem on boolean matrices*, Journ. ACM, vol. 9, no. 1, Jan. 1972, pp.11-12

HOW TO USE 1000 REGISTERS

Richard L. Sites

Department of Applied Physics and Information Science
University of California, San Diego

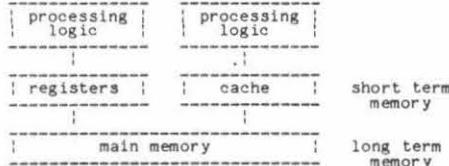
ABSTRACT

The advent of VLSI technology will allow the fabrication of complete computers plus memory on one chip. There will be an architectural challenge in the very near future to adjust to this trend by designing balanced architectures using hundreds or thousands of registers or other small blocks of memory. As the relative price of memory (vs. random logic) drops even further, the need for register-heavy architectures will become even more pronounced. In this paper, we discuss a spectrum of ways to exploit more registers in an architecture, ranging from programmer-managed cache (large numbers of explicitly-addressed registers, as in the Cray-1) to better schemes for automatically-managed cache. A combination of compiler and hardware techniques will be needed to maximize effective register use while minimizing transmission bandwidth between various memories. Discussed techniques include merging activation records at compile time, predictive cache loading, and "dribble-back" cache unloading.

I. INTRODUCTION.

VLSI technology will soon make it possible to put an entire computer plus a large number of storage locations (perhaps 100-1000 registers) on a single chip. On a larger scale of a computer occupying a few printed-circuit boards, VLSI memories will allow economical designs with a number of localized memories that are "closer" to the processor logic using them than the larger main memory (Figure 1). How can computer architects make effective use of such short-term memories?

Figure 1. A computer with two localized short-term memories:



In this paper, we first present a framework of issues for comparing short-term memory designs, then we present some new techniques for facing these issues. Our basis of comparison is a simple computer with no short-term memory, but only long-term main memory. All operations are done memory-to-memory, with no intermediate registers. Our quest is to find economical ways of adding short-term memory(ies) to this base machine.

A generalized short-term memory cell (STM cell) consists of three fields, some or all of which may be physically realized: a short name, a long name, and some data, as shown in Figure 2. In

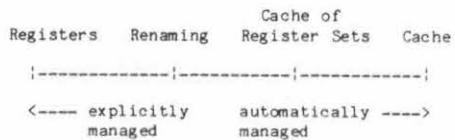
this model, a set of eight ordinary general-purpose registers would be represented by eight STMs, having short names 0-7, no long names, and one word of data each. An ordinary 1K word cache memory with 4-word lines (i.e. groups of four contiguous words) are moved into or out of the cache) would be represented by 256 STMs, each having an anonymous short name (the physical cache memory location), a main memory address for a long name, and four words of data. Other STMs will be presented below.

Figure 2. Generalized short-term memory cell (STM cell):

short name	long name	data
------------	-----------	------

Our spectrum of discourse ranges from one extreme of ordinary registers, whose use is totally explicit and completely visible to the machine-language programmer, to the other extreme of ordinary cache memory, whose use is totally automatic and intended to be invisible to the programmer. We will examine two middle-range concepts of partly-explicit, partly-automatic management, as shown in Figure 3.

Figure 3. Spectrum of discourse:



II. DESIGN ISSUES.

There are three major motivations for, and three related design issues in introducing short-term memory into a computer design:

1. Fewer address bits.
2. Faster access.
3. Lower bus bandwidth to long-term memory.
4. Load/store instructions for short-term memory.
5. Access to the most recent data.
6. Usage density of short-term memory.

Each of these ideas is briefly explained below.

Fewer address bits. If a frequently-used data item is moved from main memory to a general register, then the register number (short name) can be used to refer to that datum instead of the main memory address (long name). This results in instruction formats that are more compact than formats for our pure memory-to-memory base machine, and this compactness is a major advantage of conventional register-machine architectures. Pure

stack-machine architectures carry this concept even further by using zero instruction bits to specify the top of stack. An ordinary cache memory does not save any address bits in instruction formats.

Faster access. If a frequently-used data item is moved from main memory to a short-term memory location, then access to that datum is often speeded up by a factor of 4-20 (or even 1000 when considering main memory to be a cache on a paging drum). This faster access comes about through a combination of faster circuit technology for the short-term memory cell, shorter wire delays, less bus contention, and simpler address decoding. Small, explicitly-addressed register files will always offer slightly faster access than cache designs, since the cache must always take some time to find a match for the long name presented. Today's fastest cache designs have two-clock-cycle access, while registers often have one-cycle access.

Lower bus bandwidth. If most memory accesses are to the short-term memory, then the bandwidth needed for the long-term memory bus can be substantially lower than in our base machine. This allows a more economical design if multiple processors or I/O devices are connected to a single long-term memory, as in the PDP-11/60 design [1]. This may also allow use of a serial-multiplexed bus instead of a more expensive fully-parallel bus, thus saving wires, pins, or silicon area.

Load/store instructions. If moving data into and out of the short-term memory is done explicitly via software instructions, then two costs are suffered: first, a programmer or compiler must decide where to insert the data movement instructions; and second, instruction bits and time are consumed by these explicit commands. This overhead runs counter to the address bit savings discussed in the first point above. Cache memories neither save instruction bits nor cost data movement instructions. On the other hand, explicit register loads are easy to implement and can be positioned to pre-fetch data so that it arrives at the short-term memory just before it is needed; demand-fetch cache schemes cannot do this. Predictive cache hardware is just beginning to be investigated [2], and has recently been implemented in the Amdahl 470/V8. The address stream presented to cache can be viewed as a group of interleaved arithmetic progressions. If a simple algorithm can be used to decompose address streams into these progressions, then a cache could prefetch data in each progression.

There is another kind of load/store overhead associated with using a short-term memory: when calling a subroutine, switching tasks, or starting an I/O transfer, it is often necessary to save the current machine state, or to force it to be consistent. This means explicitly saving and restoring all the programmer-visible registers in a machine architecture, and perhaps explicitly copying a cache to main memory, or purging a translation lookaside buffer (TLB) or some otherwise "hidden" short-term memory. As short-term memories become larger and more prevalent, this load/store overhead will become a dominant speed factor. Already, machines such as the IBM 370 have introduced partial purge instructions to avoid invalidating an entire TLB of 128 entries, and the Cray-1 software has to struggle with trying not to save all $8+8+64+64+512 = 656$ registers at every subroutine call or interrupt [3].

THE ADVENT OF LARGE, CHEAP SHORT-TERM MEMORIES DEMANDS BETTER SOLUTIONS TO THE LOAD/STORE OVERHEAD PROBLEM.

Stale data. If a datum is copied to a short-term memory, and then one of the two copies is changed, subsequent access to the other copy will result in fetching stale data. This is obviously a disaster. For a hardware-managed cache memory, simple preventatives for the stale data problem involve notifying the cache of the addresses of all main memory cells changed by an processing or I/O logic in any part of a computer system. This runs counter to the lower bus bandwidth issue above. For a programmer- (or compiler-) managed register, the stale data problem is often prevented by storing the register just before some operation that might access the long-term memory copy, then reloading the short-term memory after that operation. Such operations are surprisingly frequent unless an exhaustive analysis of the program is performed. For example, if a program makes many references to one element of an array, say $A(3)$, then it is desirable to keep a copy of that element in some register. However, any other reference to the same array, such as $A(I)$, potentially accesses the third element, so the register copy of $A(3)$ must be stored before a fetch from $A(I)$, and reloaded after an assignment to $A(I)$. Depending on the language involved, it can take extensive flow analysis on the part of the compiler or programmer just to discover whether a reference to $A(I)$ is possible during the time that $A(3)$ is in a register. For example, in Fortran it is possible that $A(3)$ is kept in a register inside some loop, but the loop includes a branch to some far-away piece of program that changes $A(I)$ then branches back into the loop! If a compiler or programmer is not willing to do this sort of flow analysis, then IT IS NOT POSSIBLE TO KEEP $A(3)$ IN A REGISTER without being exposed to the stale data problem. This is the fundamental reason why simple compilers rarely make effective use of registers, and why many assembly-language programs are difficult to modify by someone other than the original author. A copy of $A(3)$ could be kept in a cache memory with no stale data problem, since the cache monitors all accessed addresses for a possible match, and supplies the most recent data if one is found.

The stale data problem also forces the saving and restoring of almost all registers across a subroutine call on register machines: continuing the above example, the subroutine might refer to $A(3)$, expecting to find the most recent value in its allocated main memory location, not in some register.

There is one more aspect to the stale data problem — aliases. If a long-term memory location can be accessed via more than one name, either because two distinct virtual memory addresses are mapped to the same physical address, or because two distinct high-level language variables in fact refer to the same location (e.g. one is a global variable, and the other is the same variable passed to a parameter), then it is possible that neither a hardware nor a software (compiler) mechanism will detect that an assignment to one name should update a copy of the other name kept in some short-term memory. In virtual memory systems, avoiding this problem involves either prohibiting aliases by software convention, or building cache hardware that compares only physical addresses, not virtual ones. In compiler systems, aliases are either detected

through extensive analysis of a program, or no copies of variables can be kept in registers across references to global variables, parameters, pointer assignments, subroutine calls, or a number of other such common occurrences.

AS SHORT-TERM REGISTER MEMORIES GET LARGER, SUBROUTINE CALLS WILL GET SLOWER, UNLESS WE FIND BETTER SOLUTIONS TO THE STALE DATA AND ALIAS PROBLEMS.

The stale data problem in all its forms is probably the hardest design problem to be faced in any system that creates copies of data. The extensive compiler analysis required to take full advantage of fast registers is one reason that cache memories have become so popular -- the hardware substitutes continual address comparisons during execution for compile-time comparisons. Thus, we have a trade-off: for simply-compiled code an N-word cache memory performs better than an N-word register memory, while for carefully-optimized code an N-word register memory performs faster (because of the inherently faster access mentioned above), and is simpler to build.

Usage density. If an architecture provides 200 words of short-term memory, but most programs use only 50 of these words, the memory is under-utilized. One "solution" in such a situation is to make the short-term memory smaller, but in the long run the opposite is preferable -- design the software to make effective use of more short-term memory. One example is in order: the Cray-1 provides 64 T-registers, each 64 bits wide with 1-cycle access (compared to 11-cycle access to a random word in main memory). To avoid load/store overhead, some system software uses none of these registers. One compiler that does generate code that uses the T-registers is the Pascal compiler at Los Alamos. It places local scalar variables into T-registers, but the short subroutines encouraged by clean Pascal coding style often have fewer than five such local variables. Thus many Pascal programs use only about 10% of the available short-term registers. For such a machine, we need software designs that use more registers. One such design is described in Section V below.

III. CACHE MEMORIES.

We will briefly summarize how ordinary cache memories fare with respect to the above six design issues. Figure 4 shows a cache memory as an STM cell associating a long name with some data.

Figure 4. Cache memory as an STM cell:

short	long	data
name	name	

Address bits. Cache memories save nothing in instruction formats.

Access time. Faster than long-term memories, but not quite as fast as registers built out of identical circuits.

Bus bandwidth. As effective as registers with the same load/store characteristics. Often cuts down bandwidth by a factor of 10 (see e.g. [1]).

Load/store overhead. No instruction overhead,

except for rare "purge the cache"-type instructions.

Stale data. The forte of cache design -- once the virtual address alias problem is dealt with, cache memories completely solve the software-level alias problem.

Usage density. This is also a strong point of cache systems -- blindly doubling the size of a cache will usually have a much better performance improvement than blindly doubling the number of equivalent registers.

IV. REGISTERS.

We will briefly summarize how ordinary register memories fare with respect to the above six design issues. Figure 5 shows a register memory as an STM cell associating a short name with some data.

Figure 5. Register memory as an STM cell:

short	long	data
name	name	

Address bits. The forte of register designs -- instruction formats shrink.

Access time. The simplicity of explicitly and directly addressed registers gives an inherent speed advantage over caches.

Bus bandwidth. Similar to cache in cutting down data accesses.

Load/store overhead. On many register machines, 25% or more of all instructions are Loads or Stores (see [1,p.351] or [4] for examples). The instruction bits for these must be balanced against the address bits saved in other instructions. Data may be pre-fetched.

Stale data. No hardware or execution time is "wasted" in trying to detect stale data, but effective use of registers demands compile-time analysis.

Usage density. Again, careful compile-time analysis is needed to take advantage of more registers. Changing assembly language code to use more registers cannot usually be done automatically.

V. TECHNIQUES FOR EFFECTIVE USE OF LARGE SHORT-TERM MEMORIES.

Notice how complementary the above two lists are (compared to our base machine):

	cache	registers
address bits	-	+
access time	+	+
bus bandwidth	+	+
load/store	+	-
stale data	+	-
usage density	+	-

Can we find some way to use the best features of both schemes? Are there techniques that are a merging of the two extremes? How can we trade-off

compile-time analysis vs. run-time analysis? Some possible solutions are discussed below.

Renaming. We can separate the idea of short names from the idea of fast access by defining a RENAME operator: RENAME X,Y means that the short name X will be used to access the long name Y until another RENAME involving the same X occurs. RENAME is like LOAD of a register in that subsequent accesses to Y can use just the short name, but it differs from LOAD because no data movement is implied. Hence, we get the short name without necessarily getting faster access. So what is the advantage of RENAME over LOAD? Figure 6 shows a RENAME mechanism as an STM cell associating a short name with a long name. A similar instruction was implemented for the index registers of the IBM 7030 (Stretch) [5].

Figure 6. Rename memory as an STM cell:

short	long	data
name	name	

First, RENAME can be implemented in conjunction with a cache memory, such that RENAME gives strong hints to the cache to load (or pre-fetch) the data at location Y. This restores the speed improvement of LOAD. Second, no explicit STORE instruction is associated with RENAME -- the use of the short name X instead of the long name Y is just discontinued at some point in a program. This saves a little instruction space, and it means that, for example, a compiler does not have to do the flow analysis to detect all branches out of a loop in order to find all the places to insert the STORE X,Y to match a LOAD X,Y at the beginning of the loop. Third and most importantly, a compiler does not have to do any alias analysis. As discussed above, when using LOAD/STORE to keep a copy of Y in register X, all other references to Y must be found and changed, or X must be appropriately restored to Y and rerefetched around any constructs that potentially touch Y. With RENAME, the implementation must ensure that references using the short name X and the long name Y both access the same actual data. Under these circumstances, use of the short name X does not require any flow analysis to find other uses or potential uses of Y.

Cache of register sets. Consider a machine with 16 general registers in its architecture. These registers are normally saved in main memory when calling a subroutine, and reloaded from main memory when returning from a subroutine. As discussed above, we desire to build machines with many more than 16 registers, but we don't want to slow down all subroutine calls. Assuming that almost all registers are in use at the point of call, and almost all will be used by the subroutine (so that we cannot avoid some sort of save/restore), then one way to speed up the call linkage is to have duplicate register sets. Say there are four sets, 0-3, and that the calling subroutine is using set 1. Then the called routine just starts using set 2, and no data movement of set 1 to main memory is needed. This makes the subroutine call quite fast, and it also makes the linkage overhead no longer proportional to the number of registers. When the subroutine returns, the machine just switches back from set 2 to set 1.

There are two flaws in the above scheme that need fixing. First is the obvious question of how

to do the fifth nested subroutine call. The answer is that after switching from register set 1 to register set 2, a cache-like mechanism is needed to dribble-back register set 1 to the place in main memory that it would have gone in the simple machine. Dribble-back means that the requisite 16 STOREs are queued at a low priority, so that whenever the running subroutine (using set 2) does not need a bus cycle to main memory, one of the queued stores is done. After the first 16 unused memory cycles pass, all of register 1 is properly stored in main memory, so more nested subroutine calls can reuse that register set. This scheme stands in stark contrast to existing machines that provide multiple register sets, such as the RCA Spectra 45 (IBM 360-like), or some models of the PDP-11, which have four register sets, but they have dedicated uses (operating system, kernel, real-time interrupt, and all user code is a typical allocation of the four), and have no automatic recycling of data to main memory.

The dribble-back technique also stands in stark contrast to the usual STORE MULTIPLE of registers at time of call, because the called subroutine need not wait until the stores finish before starting its execution. In fact, by making the priority of dribble-back stores lower than that of other stores, the register saving always uses otherwise wasted bus cycles, i.e. the register saving is completely free in terms of execution time. Since register save/restore is already a significant overhead on many machines with general registers, and since the trend is toward more registers and more short subroutines as a programming style, dribble-back will become even more significant for saving subroutine linkage time. The Amdahl 470/V6 already uses a form of dribble-back to implement the PURGE TLB instruction (which must invalidate all 128 locations in the virtual address lookaside buffer) [6]. The implementation involves a duplicate set of VALID bits for the TLB, so the PURGE TLB instruction simply switches to the other set, which has previously all been set to "invalid". During the next 128³ machine cycles, each bit of the just-used set is changed to "invalid". So long as two PURGE TLB instructions are separated by at least that many machine cycles, the implementation is extremely fast (in direct contrast to the IBM 370/168 implementation). This matches the operating system software, which only rarely executes a PURGE TLB. If a second such instruction is issued too soon, the 470/V6 CPU just waits until the previous invalidation cycle is finished.

A subroutine return can simply start using a previous register set, unless that set was dribbled-back to main memory then overwritten with registers for a more deeply nested subroutine call. In this case, the registers need to be reloaded from the data in main memory. In general, it is easy to keep a COPY bit associated with each register set, such that the COPY bit is on if the register set is an exact copy of the corresponding data in main memory. The copy bit is turned on when the last register has been dribbled-back to main memory, and it is turned off again if a nested call reuses that set. It is also turned on when the last register is reloaded from main memory. Then subroutine calls and returns can just use a register set if its COPY bit is on, and must wait for the main memory data movement to catch up if the bit is off.

Consider a deep subroutine nest of A calls B calls C calls D calls E calls F, with four sets of

registers. A uses set 0, B set 1, C set 2, D set 3, and E uses set 0 after all of A's data is dribbled-back to memory. Similarly, F uses set 1 after B's data is saved. When F returns to E, it is possible to start reloading B's data, so that when C is later ready to return to B, there will be no delay. On the other hand, if the very next instruction after F's return to E is a call from E to G, set 1 is needed for G to use, and any of B's data loaded into set 1 is wasted effort.

The thoughtful reader will have noticed that we are just running a top-of-stack buffer for a stack of register sets. For such buffers, an amount of hysteresis is useful: once a register set is stored, do not reload it immediately. Instead, wait until the probable time to reload matches the probable time until the reloaded data will be needed. In the case of nested subroutine calls above, we would like to start reloading B's registers into set 1 exactly 16 main memory cycles before C returns to B (assuming no outside access interference). In general, we cannot exactly predict when to start reloading B's data, but we can perhaps safely wait until E returns to D and D returns to C. Similarly, we could apply hysteresis at the other end of the buffer by not even starting the stores of A's registers until 16 cycles before D calls E, i.e. until just before that register set will need to be reused by a deeper subroutine.

The major effect of introducing some hysteresis along with multiple register sets is that we diminish then needed bandwidth to main memory. In fact, instead of asking for a given bus how much bandwidth must be supplied, the computer designer could ask "here is a fixed bandwidth: how much short-term memory and hysteresis must be supplied in order to exceed that bandwidth only rarely?" If we delay storing a subroutine's registers until, say, two more levels of subroutine call have been done, then we never even bother to save registers of a subroutine that only calls one level down then returns. For a software system that rarely nests calls three deep, it would be possible to run for hours without spending any time or bus bandwidth saving and restoring registers, yet the occasional call chain that is 12 deep is handled gracefully, and never with more data transfer than the simple scheme with only one register set.

A few paragraphs back, we mentioned two flaws. The second flaw is that after subroutine A calls B, but before A's registers are dribbled back to main memory, B may try to fetch from the place in main memory that is supposed to contain one of A's registers. Alternately, after A's registers are all safely copied to main memory, B may change the contents of one of those memory locations. If B then returns to A without calling anyone else, the simple description above would have A use the stale data in its register set, without ever reloading the changed word in main memory. The solution to this flaw involves using standard cache techniques: the unused register sets that contain copies of main memory data are exactly cache locations, and all accesses to the corresponding main memory locations must update the cache also. Thus, four register sets look like a four-line cache, with a main memory address tag associated with each line (register set), and with an associative lookup of these four tags whenever main memory is referenced. This scheme effectively ties together the two ends of our short-term memory spectrum.

VI. CONCLUSIONS AND FUTURE RESEARCH.

Registers are simple to build, fast, and small numbers of them are easy for programmers and optimizing compilers to use effectively. Cache memories are more complicated, but easier to use. Providing many registers is an attractive way for the hardware designer to use VLSI technology to support economical short-term memory. Providing a combination of hardware alias resolution and stale data prevention via cache-like address comparisons, along with many registers, may be the best total-system design for effective use of 1000 or more register locations.

Cached register sets are particularly attractive for implementing fast subroutine calls, but the same ideas also apply to implementation of hardware stacks or queues (contrast the Burroughs 7700, with 32 top-of-stack buffer registers, the automatic saving and restoring of which significantly slows down subroutine calls [7]), and to the implementation of task switching. In the latter case, complete duplicate machine states could be kept in multiple register sets.

One line of future research is to measure existing software to discover how much short-term memory hardware would be useful, and what are the parameters for managing it (for example, carefully gathered statistics on dynamic subroutine call/return activity could help decide an optimum number of register sets, plus the parameters of the hysteresis algorithm).

A second line of research is just the converse -- given a fixed arbitrary amount of short-term memory hardware, how can software be automatically re-done to take full advantage of that amount? If only a few levels of subroutine nesting can be handled quickly, automatic insertion of subroutine code inline at the point of call would decrease the number of levels of call in a software package. If many subroutines have only 5 local variables available for short-term storage and a machine provides 30 short-term registers, then a compile-time mapping of the local variables from six subroutines into one merged activation record [3] could provide a much better match to the machine -- the usage density goes way up, and calls between subroutines in such a group would not need to save or restore registers at all: each subroutine just uses a different five of the 30 registers.

Both lines of research must be pursued simultaneously if we are to take full advantage of the short-term memory architectures that VLSI technology makes economical.

REFERENCES

- [1] C.G. Bell, J.C. Mudge, and J.E. McNamara, Computer Engineering, chapter 13, Digital Press, Bedford MA, 1978.
- [2] A.J. Smith, "Sequential Program Prefetching in Memory Hierarchies", IEEE Computer, December 1978, pp. 7-21.
- [3] R.L. Sites, "An Analysis of the Cray-1 Computer", Fifth annual symposium on Computer Architecture, April, 1978, pp 101-106.

- [4] R.P. Blake, "Exploring a Stack Architecture",
IEEE Computer, May 1977, pp. 30-38.
- [5] IBM Corp., Reference Manual: 7030 Data Processing System, form A22-6530, 1960.
- [6] Amdahl Corp., Amdahl 470/V6 Hardware Reference Manual, 1976.
- [7] E.I. Organick, Computer System Organization, Academic Press, New York NY, 1973, p. 91, 101.

