

# PROCEEDINGS

*of the First Annual*

# SYMPOSIUM ON

# COMPUTER

# ARCHITECTURE

*Edited by:*

G. J. LIPOVSKI  
S. A. SZYGENDA



COMPUTER ARCHITECTURE NEWS, DECEMBER 1973, VOL. 2, NO. 4

IEEE CATALOG NO. 73CH0824-3C

CENTER FOR INFORMATICS RESEARCH TR-73-107



UNIVERSITY OF FLORIDA  
DECEMBER 9-11, 1973

GENERAL CHAIRMAN  
G. JACK LIPOVSKI

PROGRAM CHAIRMAN  
STEVE A. SZYGENDA

PROGRAM COMMITTEE

Al Avizienis	Don Gibson
Gordon Bell	Al Hoagland
Harvey Cragon	Dave Rouse
Jack Dennis	Harold Stone
Mike Flynn	Bruce Wald
Oscar Garcia	

SYMPOSIUM COMMITTEE

Wayne Chen	Bill Kaiser
Oscar Garcia	Stanley Su
Gil Hansen	Julius Tou
George Haynam	Ken Watson

*Co-Sponsors:*

ACM SIGARCH  
Center for Informatics Research, University of Florida  
Computer Society of the IEEE

# PROCEEDINGS OF THE FIRST ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE

*Edited by:*

G. J. LIPOVSKI  
S. A. SZYGENDA

Copyright © 1973 by:  
**The Institute of Electrical and Electronics Engineers, Inc.**  
345 East 47th St., New York, N. Y. 10017  
**Association for Computing Machinery, Inc.**  
1133 Avenue of the Americas, New York, N. Y. 10036

University of Florida  
Gainesville

# PREFACE

This symposium may well be, in the hind-sight of ten years from now, a marked turning point in Computer Architecture. With the dissolution of the Spring and Fall Joint Computer Conferences, one of the major forums for Computer Architecture has been lost. So we have begun an annual symposium on Computer Architecture, to be rotated from year to year throughout the world. The atmosphere of such a symposium should be more suitable for the professional interchange of ideas than is possible at a large conference. Indeed, from the quality of papers that have been submitted to this symposium, it is clear that the time is here for a top quality symposium. We are pleased to say that the papers in this symposium are those that at least two reviewers rated in the top category. We are sorry that, because of this, a large number of very good papers were rejected. However, we have passed these papers, together with their reviews, on to editors of journals that cover Computer Architecture, for their consideration. We feel that, to encourage the submission of good papers to a symposium, it is desirable for us to send those papers that don't happen to fit into a session, but are very good papers, to journals for further reviewing.

The papers in the symposium indicate the growth of Computer Architecture as a science. Although it is difficult to explain the reasoning behind the decisions made in an architecture, in particular, the architecture of a practical machine, this reasoning is the basis of a science. It is too easy to simply show the final master-piece, as an artist would do. This is the "Moses Complex", as we call it, where the architecture of a practical machine is presented as if it is burned into stone, and need not be questioned. Several papers in this conference are directed at the reasoning process itself. We intend to encourage other authors to focus on reasons for the architecture by having an open panel discussion at the end of each section. We hope that the attendees will emphasize questions on the reasoning behind the architecture, and the authors will prepare for such questions. If this becomes a tradition in this annual symposium, it should orient authors toward the scientific explanation of their architectures for later symposia.

Parallel to this emphasis on explaining the reasoning, a number of papers in the symposium are on description languages. We believe that a widely used description language will permit the compression of detail so that all of the essential information is all there, but does not fill up a large part of the paper. We believe that the development of a good description language is another cornerstone to the growth of Computer Architecture as a science.

There is a wide interest, as exemplified in several papers, in the pedagogy of Computer Architecture. These papers show the need for courses which abstract the principles of Computer Architecture. There is also a trend to introduce more laboratory experience into Computer Architecture, to balance the thrust towards principles with a tie to the reality of hardware.

A survey of the session titles shows some of the other exciting areas of current research. Some of the traditional areas, such as the design of fast arithmetic units, have been rather thoroughly researched, although some questions are yet unresolved. The current areas that are receiving particular attention are the connection of modular systems and fault tolerant or

fail-soft processing systems. As a special case of modular systems, pipeline and cellular systems are receiving continued attention. The growth of LSI, and the advent of microcomputers in particular, is evoking considerable excitement in modular systems of all kinds. There are indications that modularity of various kinds will provide some useful tools in making computers fault tolerant or fail-soft. A while back, someone wrote that in the next couple of decades, Computer Architecture will not change the computers that will be built, that they will differ from present computers in that they are faster or have more primary memory, and so on. I cannot agree! Driven by the user's demands for fault tolerant computing and the change in technology towards the use of microcomputers, Computer Architecture will have considerable impact on the machines that are going to be built over the next decade.

This symposium owes a great deal to a number of people, whom I wish to recognize. Mike Flynn deserves our gratitude as the chairman of TCCA and SIGARCH who initiated this symposium. We are no less appreciative of the help of the current chairman of TCCA, Harold Stone, and the current chairman of SIGARCH, Chuck Casale. Steve Szygenda has done an excellent job, together with his Program Committee, of attracting and reviewing papers for the symposium. The Program Committee deserves our deepest gratitude. They are:

Al Avizienis	Don Gibson
Gordon Bell	Al Hoagland
Harvey Cragon	Dave Rouse
Jack Dennis	Harold Stone
Mike Flynn	Bruce Wald
Oscar Garcia	

I also wish to thank the members of the Symposium Committee, who have helped me set up the symposium. They are:

Wayne Chen	Bill Kaiser
Oscar Garcia	Stanley Su
Gil Hansen	Julius Tou
George Haynam	Ken Watson

We wish to thank the Department of Electrical Engineering, and its acting chairman, Gene Chenette, for the extensive use of its facilities, and the Center for Informatics Research, directed by Julius Tou, for his guidance and assistance. We are grateful for the help of the Engineering Publications office, under Dick Dale, for their assistance in preparing the call for papers and advance program, and for Storter Printing for printing these fliers and the proceedings. Finally, every conference is made to work by the unselfish assistance of the secretaries. I particularly want to thank Beth Beville for her conscientious and competent assistance. We owe all of these people a great deal because, without their help, the symposium would not have been possible.

Chairman of the Symposium



G. Jack Lipovski

Center for Informatics Research

# CONTENTS

PAGE

"Markov Chain Models for Analyzing Memory Interference in Multiprocessor Computer Systems", Dileep P. Bhandarkar and Samuel H. Fuller, Carnegie-Mellon University . . . . .	1
"Interconnecting A Distributed Processor System for Avionics", George A. Anderson, Honeywell, Minneapolis . . . . .	11
"Banyan Networks for Partitioning Multiprocessor Systems", Rodney Goke, G. J. Lipovski, University of Florida . . . . .	21
"Structure of Digital System Description Languages", Harry F. Jordan and Burton J. Smith, University of Colorado . . . . .	31
"VDL - A Definitional System for All Levels", John A. N. Lee, University of Massachusetts . . . . .	41
"A Methodology for Parallel Processing Design Tradeoffs", Charles H. Radoy, George P. Copeland, Jr., and G. J. Lipovski, University of Florida . . . . .	51
"DAP - A Distributed Array Processor", S.F. Reddaway, International Computers Limited . . . . .	61
"Maximal Rate Pipelined Solutions to Recurrence Problems", Peter M. Kogge, IBM, Owego . . . . .	71
"Comments on Capabilities, Limitations and 'Correctness' of Petri Nets", Tilak Agerwala and Mike Flynn, John Hopkins University . . . . .	81
"Flowware -- A Flow Charting Procedure to Describe Digital Networks", Wayne E. Omohundro, BTL and James H. Tracey, University of Missouri . . . . .	91
"Automated Exploration of the Design Space for Register Transfer (RT) Systems", M. R. Barbacci and D. P. Siewiorek, Carnegie-Mellon University . . . . .	101
"Implementation Aspects of the Symbol Hardware Compiler", T. A. Laliotis, Fairchild Systems, Palo Alto . . . . .	111
"The Architecture of CASSM: A Cellular System for Non-numeric Processing", George P. Copeland, Jr., G. J. Lipovski and Stanley Y. W. Su, University of Florida . . . . .	121
"Deriving Design Guidelines for Diagnosable Computer Systems", John M. Hemphill, USAF and S. A. Szygenda, University of Texas, Austin . . . . .	131
"Design of Fault-Tolerant Associative Processors", Behrooz Parhami and Algirdas Avizienis, UCLA . . . . .	141
"A Fault Tolerant Multiprocessor Architecture for Real Time Control Applications", M. A. Fischler and O. Firschein, Lockheed, Palo Alto . . . . .	151
"A Varistructured Fail-soft Cellular Computer", G. J. Lipovski, University of Florida . . . . .	161
"A Hardware Laboratory for Computer Architecture Research", Jean Vaucher, Christian Rey, Universite de Montreal . . . . .	171
"Simulation Exercises for Computer Architecture Education", P. J. Knoke, Radiation, Inc., Florida . . . . .	181
"Computer Architecture Courses in Electrical Engineering Departments", M. E. Sloan, Michigan Technological University . . . . .	191
"Increasing Hardware Complexity - A Challenge to Computer Architecture Education", R. Hartenstein, Karlsruhe University . . . . .	201
"Review of the Workshop on Computer Architecture Education", George Rossmann, Palyn, Inc., . . . . .	211
"Micromodules: Microprogrammable Building Blocks for Hardware Development", Richard G. Cooper, National Security Agency . . . . .	221
"Computer Modules: An Architecture for Large Digital Modules", S. H. Fuller, D. P. Siewiorek and R. J. Swan, Carnegie-Mellon University . . . . .	231

	PAGE
"A Microprogrammed Architecture for Front End Processing", Rodnay Zaks, Universite de Technologie de Compiegne, France . . . . .	241
"Design of a Fully Variable - Length Structured Minicomputer", Z. G. Vranesic, V. C. Hamacher, and Y. Y. Leung, University of Toronto . . . . .	251
"HAPPE Honeywell Associative Parallel Processing Ensemble", Orin E. Marvel, 13964 Wildwood Drive, Largo . . . . .	261
"A Computer Architecture and its Programming Language", Mario R. Schaffner, MIT . . . . .	271

The page numbers in this Proceedings will use the following format. Page 253, will be page 3 of Paper 25. This will leave gaps in the sequence of pages, but enables us to coalate and prepare the Proceedings more quickly.

# MARKOV CHAIN MODELS FOR ANALYZING MEMORY INTERFERENCE IN MULTIPROCESSOR COMPUTER SYSTEMS<sup>1</sup>

Dileep P. Bhandarkar<sup>2</sup>  
 Samuel H. Fuller  
 Carnegie-Mellon University  
 Pittsburgh, Pennsylvania

## ABSTRACT

This paper discusses various analytical techniques for studying the extent of memory interference in a multiprocessor system with a crosspoint switch for processor-memory communication. Processor behavior is simplified to an ordered sequence of a memory request followed by an interval of processing time. The system is assumed to be bus bound; in other words, by the time the processor-memory bus completes servicing a processor's request the processor is ready to initiate another request and the memory module is ready to accept another request. The techniques discussed include discrete and continuous time Markov chain models as well as several approximate analytic methods.

## 1. INTRODUCTION

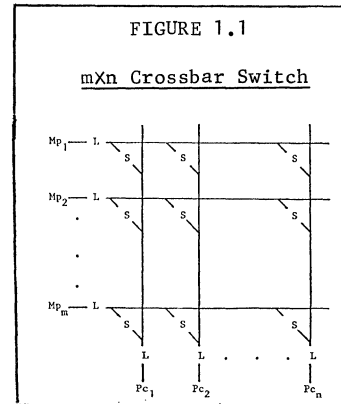
Carnegie-Mellon University is currently in the process of constructing a multiprocessor computer system (C.mmp) that will have up to 16 central processors (Pc's)<sup>3</sup> sharing the same physical address space (4) and concern has been expressed about the performance of such a system with these many active processors. In addition to the processors, there is a set of memory modules that are able to operate independently; little would be gained if all the processors had to wait for service from a single memory module. Between the processors and the memory modules (Mp's) is a  $n$  by  $m$  switch. There are a number of ways of implementing the switch, but C.mmp employs a full  $n$  by  $m$  crosspoint switch as shown in Figure 1.1. Other multiprocessors, although limited to a smaller number of Pc's, also basically use a crosspoint switch, e.g. the Burroughs D825 and the Univac 1110. For further discussion of crosspoint switches, and a variety of other switching structures, see Bell and Newell (3).

Mathematical models of computer systems can be developed at various levels of abstraction. A large number of models for time-sharing systems consider a job as a basic unit (cf. 10), and in many models of multiprogrammed computer systems the block of instructions between I/O operations is taken as a basic unit (cf. 5). However, in this study a much more detailed

<sup>1</sup>This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

<sup>2</sup>D. P. Bhandarkar is now with Texas Instruments Inc., Dallas, Texas.

<sup>3</sup>We use the PMS notation of Bell and Newell (3) in this report to describe hardware organization.



model is used to analyze interference as processors access individual words from the memory modules. Each processor's performance is measured by the number of memory accesses per unit time. The major contribution of this paper is a systematic method for a discrete Markov chain model. Other techniques described include Strecker's approximation (13), systems with exponentially distributed memory service time, and a diffusion approximation.

## 2. GENERAL MODELING ASSUMPTIONS

Due to the complexity of the problem, the exact detailed behavior of memory interference in a multiprocessor system is difficult to model. We make the following assumptions with respect to the parameters that characterize the behavior of a Pc.

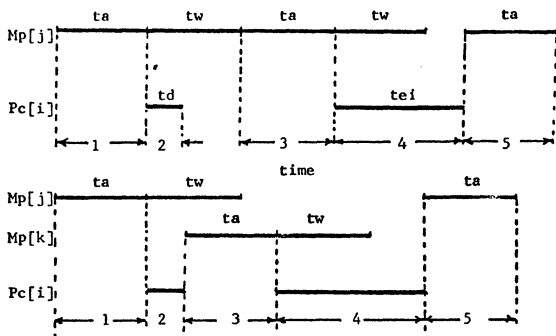
Instruction mix: In general, processor behavior varies for different instructions. However, in this paper differences in instructions are ignored. Processor behavior is modeled as an ordered sequence of a memory request followed by an interval of execution time. At this level of abstraction no distinction is made between the processing needed to decode an instruction and the processing corresponding to its execution. Thus, the processing time characterizing a Pc depicts only the aggregate behavior of the real Pc. Figure 2.1 depicts the actual and abstracted behaviors.

Processing time of Pc: The models discussed here assume that the multiprocessor systems are bus bound, i.e. the Pc is ready to initiate the next request and the Mp module is ready to accept the next request at the time the Pc-Mp bus recovers from the current access. The analysis is also applicable to multiprocessor systems in which the effective processing time,  $t_p$ , is equal to the memory rewrite time,  $t_w$ .

Access pattern of a Pc: This is the sequence of memory locations accessed by the Pc. In this study serial correlation between successive memory accesses

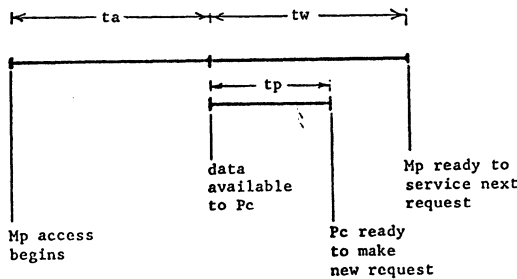
FIGURE 2.1

a. An Example of the Timing of a Typical Instruction



- Legend:
- 1 instruction fetch
  - 2 instruction decoding
  - 3 operand fetch
  - 4 instruction execution
  - 5 next instruction fetch
  - ta memory access time
  - tw memory restore time
  - td instruction decode time
  - tei processor execution time

b. Simplified Processor Behavior. Two such cycles model the instruction shown in Figure 2.1a.



will be ignored. Demand patterns will be modeled as sequences of Bernoulli trials. Memory accesses will be characterized by the memory units to which they are addressed.

**Primary memory behavior:** Memory performance is a function of the fabrication technology, i.e. core or semiconductor. It can be characterized by the access time (ta), rewrite time (tw), and cycle time (tc). Nominally, the cycle time is the sum of the other two. In this study, no distinction is made between read and write operations.

3. CONTINUOUS TIME MARKOV CHAIN MODEL

Consider a multiprocessor system which consists of n Pc's and m Mp's connected by a single crosspoint switch. Let  $P_{ij}$  denote the probability that the i-th processor requests service from the j-th memory unit. A processor is queued if it is waiting for or in the process of receiving memory service and it is active if it is currently being serviced by a memory. Likewise, a memory is said to be occupied or busy if there is at least one processor queued for that memory unit.

In this first model, we apply the classic simplifying assumption in queueing theory: we model the service time, or cycle time, of the memory modules as exponentially distributed random variables. Clearly most memory systems do not have an exponentially distributed cycle time. However, techniques such as interleaving, cache memories, and the type of memory access (read, write, read-modify-write) suggest that this exponential

assumption may be as good an approximation as the assumption that the memory cycle time is constant. Without further assumptions or approximations, we can use the results of Jackson (7), and Gordon and Newell (6), to find the performance of the multiprocessor system. This technique is also used by McCredie (9) for multiprocessors with  $tp > tw$ .

Let the number of service centers be m. The states of the system are m-dimensional vectors with non-negative integer components, the j-th component representing the queue length at center j. If  $\vec{K} = (k_1, k_2, \dots, k_m)$  is a state vector, then  $S(\vec{K}) = \sum_{i=1}^m k_i$ . Transition from one center to another is characterized by a routing probability  $R_{ij}$ , i.e. the probability of going to center j on completion of service at center i. Jackson (7) has obtained the equilibrium joint probability distribution of queue lengths for a broad class of queueing-theoretical models representing a network of service centers. Customer arrivals are modeled as a generalized Poisson process whose mean arrival rate varies almost arbitrarily with the total number of customers already in the system. Service completions at each center are also modeled as generalized Poisson processes, the mean service rate,  $\mu$ , at each center varying arbitrarily with the queue length there.

For closed queueing systems, Jackson's formulae reduces to

$$P(\vec{K}) = w'(\vec{K}) / T'(S(\vec{K}))$$

where

$$w'(\vec{K}) = \prod_{j=1}^m \frac{e^{(j)} k_j}{\mu}$$

$$\text{where } e^{(j)} = \sum_{i=1}^m e^{(i)} R_{ij}, \quad j \in [1, m]$$

$$T'(\vec{K}) = \sum w'(\vec{K}) \text{ summed over all } \vec{K} \text{ with } S(\vec{K}) = n.$$

But, with Pc requests distributed uniformly and with the bus-bound situation, or  $tp=tw$ , Jackson's model simplifies to m servers with customers circulating with uniform routing probabilities, i.e.  $R_{ij} = P_{ij} = 1/m$ . Using the above formulae we get,

$$w(\vec{K}) = \left(\frac{1}{\mu}\right)^n$$

$$T(\vec{K}) = \binom{n+m-1}{m-1} \left(\frac{1}{\mu}\right)^n$$

$$P(\vec{K}) = \left[ \binom{n+m-1}{m-1} \right]^{-1} \text{ for all } \vec{K} \text{ such that } \sum_{i=1}^m k_i = n,$$

i.e. all the states of the system are equally likely. Physically, this indicates that states with greater congestion in the queues are as likely as evenly distributed queues. The probability that a particular Mp module is idle,  $\Pr\{Mp[i] \text{ is idle}\}$ , is the fraction of the total number of states that has  $k_i = 0$ . In other words,

$$\text{Prob}\{Mp[i] \text{ is idle}\} =$$

$$\frac{\text{number of ways of assigning } n \text{ Pc's to } m-1 \text{ Mp's}}{\text{number of ways of assigning } n \text{ Pc's to } m \text{ Mp's}}$$

$$= \frac{\binom{n+m-2}{m-2}}{\binom{n+m-1}{m-1}} = 1 - \frac{n}{n+m-1}$$



$$E\{\text{number of busy Mp's}\} = \sum_{i=1}^m \Pr\{\text{Mp}[i] \text{ is busy}\}$$

$$= m^*n / (m+n-1)$$

The above expression has a number of interesting properties: the expression is symmetric in  $m$  and  $n$ ; it has a basic hyperbolic form, asymptotic to  $n$  as  $m$  gets large; and, if we let  $m=n$  the above expression becomes  $n/(2-1/n)$  and

$$\lim_{n \rightarrow \infty} E\{\text{number of busy Mp's}\} \rightarrow n/2.$$

The final observation has important implications. It states that as multiprocessor systems grow to include more and more Pc's, we are not faced with a law of diminishing returns: no matter how many Pc's are used, if we have the same number of memory modules we can expect half the processors to be active.

#### 4. A SIMPLE DISCRETE MARKOV CHAIN MODEL

For this analysis let us assume that all the Pc's are characterized by a single constant processing time  $t_p$ . Also, all the memory units are assumed to have the same cycle time  $t_c$  and access time  $t_a$ . Thus, the memory rewrite time is given by  $t_w = t_c - t_a$ . If  $t_p = t_w$  then all memory units can be considered to be operating synchronously. Thus, during any memory cycle the number of active Pc's is equal to the number of busy Mp's.

In this section a simple Markov Chain analysis is presented for the case in which the processors request every memory with equal likelihood. The state of the multiprocessor system is defined by a  $m$ -tuple where  $\sum_{i=1}^m k_i = n$  and  $0 \leq k_i \leq n$  for all  $i$ . The number of distinct

states of the system is given by the combination,  $\binom{n+m-1}{m-1}$  i.e. the number of ways in which  $n$  balls can be assigned to  $m$  bins (4). However, since all the processors behave identically, a number of the distinct states are equivalent, i.e. they have the same occupancy and have the same components, e.g. states  $(2,1,1)$ ,  $(1,2,1)$ ,  $(1,1,2)$  are equally likely. Thus, the reduced states are given by the different ways in which the number  $n$  can be partitioned into  $m$  parts. The number of such partitions (for  $n \leq m$ ) is asymptotic to

$$\frac{1}{4\pi\sqrt{3}} e^{\pi\sqrt{2n/3}} \quad (\text{cf. } 2)$$

Let the representative state  $S_i$  denote the set of compositions of the number  $n$  that yield the same partition, e.g. the compositions  $(2,1,1)$ ,  $(1,2,1)$  and  $(1,1,2)$  correspond to the partition of the number 4 which has two 1's and one 2. Further, let  $S_{i,j}$  be the individual compositions of the partition typified by representative state  $S_i$  and  $S_{i,j}$  be that composition which has its components arranged in monotonic non-decreasing order, i.e.  $(2,1,1)$  for the above example.

Let  $X_{ij}$  denote the probability of a transition from  $S_i$  to  $S_j$ . Then, due to the symmetry of the problem,

$$X_{i,j} = \sum_{S_{j,k} \in S_j} \{ \text{Transition from } S_{j,k} \text{ to } S_{i,k} \}$$

Let the  $m$ -tuple  $(k_1, k_2, \dots, k_m)$  denote the state of the Markov chain. If  $x$  is the number of non-zero elements in this vector then at the end of the memory cycle,  $x$  new processors have to be reassigned to memory

modules. At the end of the current memory cycle the queue is characterized by the  $m$ -tuple  $(j_1, j_2, \dots, j_m)$ , where

$$j_i = \begin{cases} k_i - 1 & \text{if } k_i > 0 \\ 0 & \text{if } k_i = 0 \end{cases}$$

A new state  $(l_1, l_2, \dots, l_m)$  is reachable from  $(k_1, k_2, \dots, k_m)$  if and only if  $l_i \geq j_i$  for  $1 \leq i \leq m$ . If the above condition is satisfied the probability of the state transition is given by

$$\binom{x}{d_1} \binom{x-d_1}{d_2} \binom{x-d_1-d_2}{d_3} \dots \binom{x-\sum_{i=1}^{m-1} d_i}{d_m}$$

$$\text{where } d_i = l_i - j_i$$

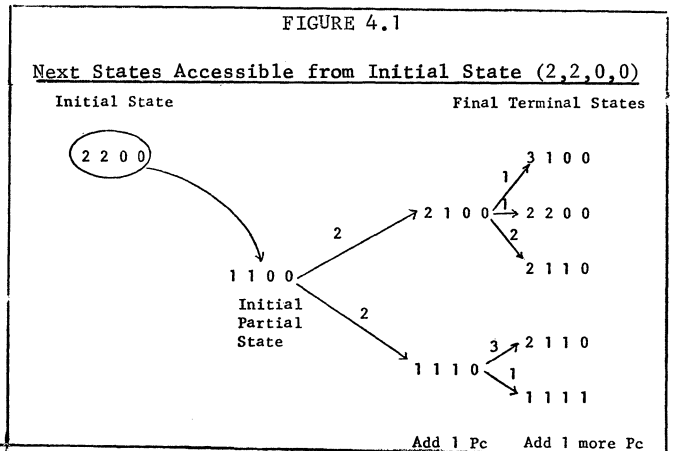
$$\text{i.e. } \frac{x!}{d_1! d_2! \dots d_m!} * \left(\frac{1}{m}\right)^x$$

Note that since  $\sum_{i=1}^m k_i = \sum_{i=1}^m l_i = n$ ,  $\sum_{i=1}^m d_i = x$ .

Thus, we now have a formula for generating the transition probabilities. Due to the symmetry of the problem it suffices to generate only the transition probabilities for the representative class of states. All the different ways of obtaining the same partition are lumped together to form a reduced state.

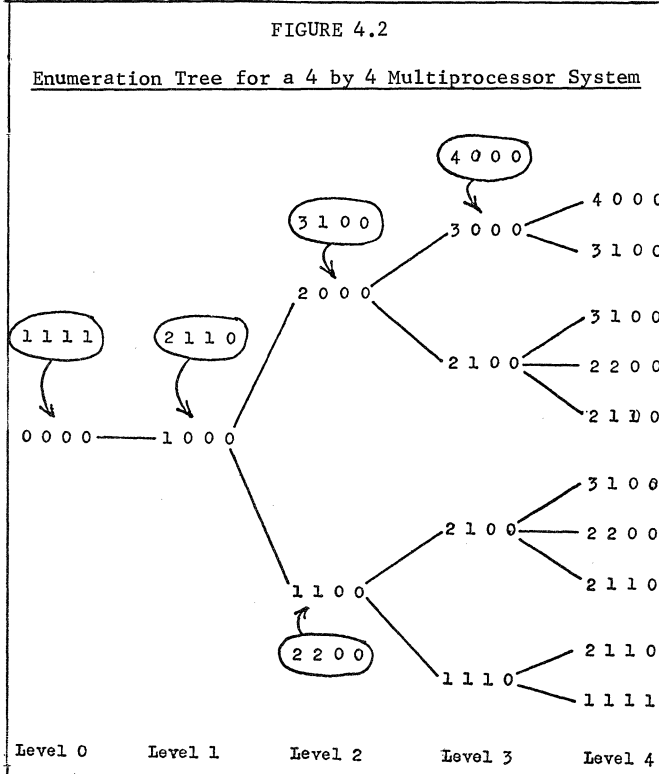
To illustrate a computational method<sup>1</sup> for generating the transition probabilities consider an example of a 4 by 4 system. The number 4 can be partitioned in five different ways:  $\{(4,0,0,0); (3,1,0,0); (2,2,0,0); (2,1,1,0); (1,1,1,1)\}$ .

These partitions represent five equivalence classes that characterize the state of the Markov Chain. Let us consider the state  $(2,2,0,0)$ . At the end of a memory cycle, the resultant partial state is  $(1,1,0,0)$  with two free processors to be reassigned. Figure 4.1 shows the different ways in which these two Pc's can be assigned, one at a time, to reach a new partial representative state. After both Pc's are assigned a terminal state is reached. The number on the arrow indicates the number of ways of reaching the partial or terminal state that the arrow points to. Now the number of ways in which a final state can be reached from the initial state can be computed by traversing the tree, e.g. there are  $2 \times 1$  ways of reaching  $(1,1,1,1)$  and  $(2 \times 2 + 2 \times 3)$  ways of reaching  $(2,1,1,0)$  from  $(2,2,0,0)$ .



<sup>1</sup>The use of a tree to generate the transition probabilities was suggested by F. Baskett and D. Chewning of Stanford University.

It is possible to construct a single tree with different pointers for different initial states. Figure 4.2 shows a complete tree for a 4x4 system. Initial states are circled. The entire transition matrix can be generated by traversing this tree. A convenient way of traversing this tree is by using a stack which has depth equal to one more than the number of Pc's. At each level the stack contains a partial state and has a pointer to the initial representative state (if any) from which it is derived. The stack is initialized to contain the path that leads to the top-most final state. For this example the transition matrix is shown in Figure 4.3.



The following theorems can be used to increase the efficiency of the program that generates the transition probabilities.

**Theorem 1.** There is a one-to-one correspondence between a representative state and a partial state that the representative state reduces to at the end of a cycle.

**Proof.** Let  $(k_1, \dots, k_m)$  be a representative state. The partial state at the end of the cycle is given by  $(j_1, j_2, \dots, j_m)$  where

$$j_i = \begin{cases} k_i - 1 & \text{if } k_i > 0 \\ 0 & \text{if } k_i = 0 \end{cases}$$

Since no two representative states are alike and  $\sum_{i=1}^m k_i = n$ , it follows that the partial states are distinct. ■

**Theorem 2.** A partial state at level L in the enumerative tree of Figure 4.3 can correspond to a terminal state with exactly n-L occupied Mp's.

<sup>1</sup> For an alternative method for traversing the tree see [1].

FIGURE 4.3

Steps in the Generation of the Transition Matrix

	4 0 0 0	3 1 0 0	2 2 0 0	2 1 1 0	1 1 1 1
4 0 0 0	1	1	0	1	4
3 1 0 0	3	3+3	2	3+3+6	12+12+24
2 2 0 0	0	3	2	3+6	12+24
2 1 1 0	0	6	4+6	6+12+18	24+48+72
1 1 1 1	0	0	2	6	24

**STEP 1:**  $\hat{X}_{ij}$  is the number of ways of reaching i from j.

**STEP 2:**  $X_{ij} = \frac{\hat{X}_{ij}}{\sum_i \hat{X}_{ij}}$  (Note that  $\sum_i \hat{X}_{ij} = m^x$ , where x of the components of j are non-zero)

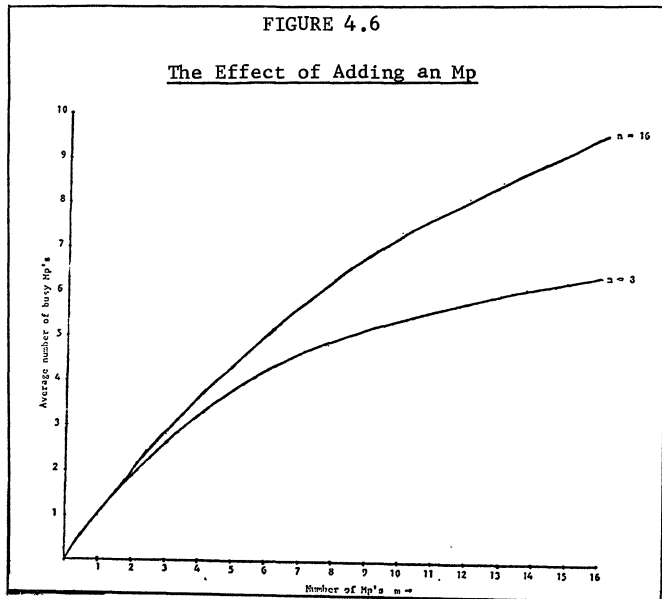
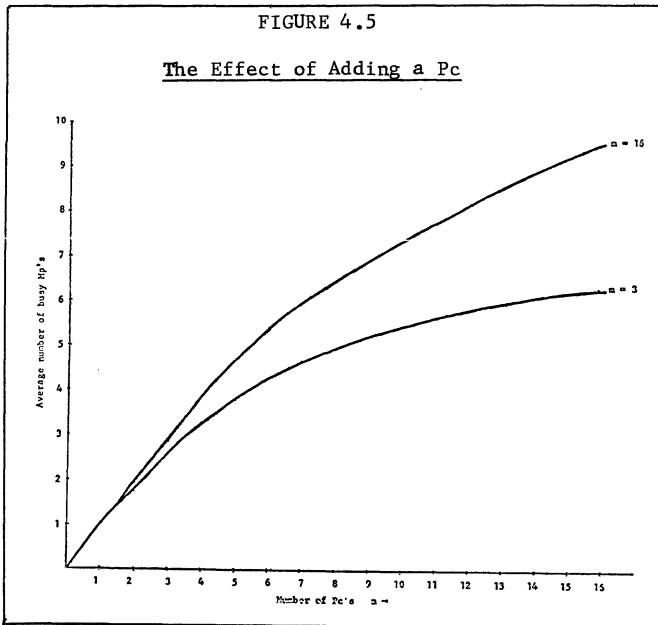
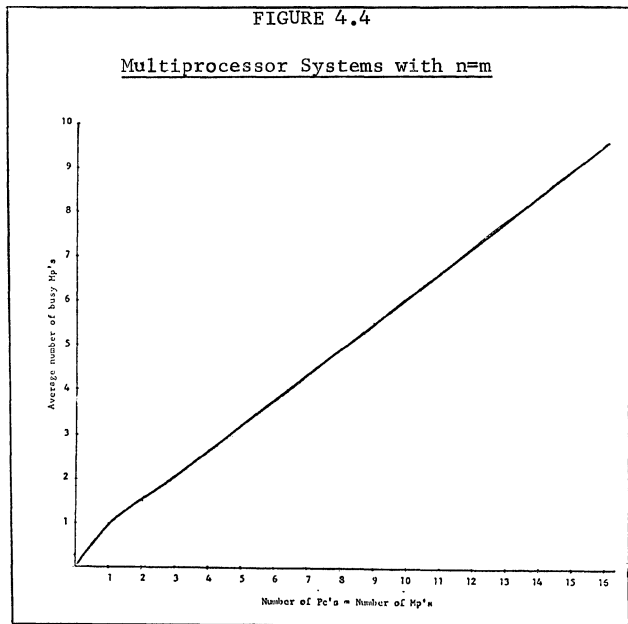
Final equations to be solved simultaneously :

$$\begin{bmatrix} P_{4000} \\ P_{3100} \\ P_{2200} \\ P_{2110} \\ P_{1111} \end{bmatrix} = \begin{bmatrix} 0.25 & 0.0625 & 0.000 & 0.015625 & 0.015265 \\ 0.75 & 0.3750 & 0.125 & 0.187500 & 0.187500 \\ 0.00 & 0.1875 & 0.125 & 0.140625 & 0.140625 \\ 0.00 & 0.3750 & 0.625 & 0.562500 & 0.562500 \\ 0.00 & 0.0000 & 0.125 & 0.093750 & 0.093750 \end{bmatrix} \begin{bmatrix} P_{4000} \\ P_{3100} \\ P_{2200} \\ P_{2110} \\ P_{1111} \end{bmatrix}$$

SUBJECT TO  $P_{4000} + P_{3100} + P_{2200} + P_{2100} + P_{1111} = 1$

**Proof.** Let  $\vec{J} = (j_1, j_2, \dots, j_m)$  be a partial state in the tree depicted in Figure 4.2. Furthermore, let the number of non-zero elements in the partial state be y and let  $\sum_{i=1}^m j_i = n-x$ . Since one Pc is always removed from a non-empty queue at the end of a cycle,  $\vec{J}$  is a partial state that can be reduced from a valid representative state  $\vec{K} = (k_1, k_2, \dots, k_m)$ , if and only if the number of non-zero elements in  $\vec{K}$  is x, and  $x \geq y$ . Note that x and y are both less than or equal to  $\min(m, n)$  and  $\sum_{i=1}^m k_i = n$ . If  $x < y$  then there is no representative state  $\vec{K}$  that corresponds to the partial state  $\vec{J}$ . If  $x \geq y$ , then the representative state is obtained by adding y 1's to the non-zero elements of  $\vec{J}$  and replacing x-y zeros of  $\vec{J}$  by 1. At level L,  $\sum_{i=1}^m j_i = L$ . Therefore, x, the number of occupied Mp's in  $\vec{K}$ , is equal to n-L. ■

Figure 4.4 shows the average number of busy Mp's when  $n=m$ . The curve has an almost constant slope of .586 for  $n > 4$ . Figures 4.5 and 4.6 show the effect of adding a Pc and an Mp respectively on the average number of busy Mp's.



## 5. APPROXIMATIONS

Strecker's Approximation. Strecker (13) has an approximate closed form solution to the discrete Markov Chain model presented here. His approach is equivalent to removing the queued processors from all the memory modules at the end of a memory cycle and reassigning them. Thus the state of the system is considered independent of the state during the last cycle. If we use this assumption the distribution of Pc's queued for an Mp follows the binomial distribution:

$$\Pr\{Y=r\} = \binom{n}{r} \left(\frac{1}{m}\right)^r \left(1 - \frac{1}{m}\right)^{n-r}$$

where Y is a random variable equal to the number of Pc's queued for Mp[j] and  $p_{ij} = \frac{1}{m}$  for all i and j. Thus,

$$\begin{aligned} \Pr\{\text{Mp}[j] \text{ is busy}\} &= 1 - \Pr\{\text{Mp}[j] \text{ is idle}\} \\ &= 1 - \left(1 - \frac{1}{m}\right)^n \end{aligned}$$

In other words, the occupancy of Mp[j] is  $1 - \left(1 - \frac{1}{m}\right)^n$ , and

$$\begin{aligned} E\{\text{no. of occupied Mp's}\} &= \sum_{j=1}^m \Pr\{\text{Mp}[j] \text{ is busy}\} \\ &= m \left[1 - \left(1 - \frac{1}{m}\right)^n\right] \end{aligned}$$

Strecker's approximation overestimates the unit execution rate, but it is encouraging to note that such a simple expression is within 6 to 8% of the exact solution of the Markov Chain model for  $m/n > 0.75$ . Moreover, the expression  $m \left[1 - \left(1 - \frac{1}{m}\right)^n\right]$  can be written in an exponential form as

$$m \left\{1 - \exp\left[n \ln\left(1 - \frac{1}{m}\right)\right]\right\}$$

and the relaxation time,  $\left[\ln\left(1 - \frac{1}{m}\right)\right]^{-1}$ , approaches m as m gets large.

Diffusion Approximations. An approximation method that has been proposed for the solution of general queueing networks is the diffusion approximation (cf. 8,11). A discrete-state process is approximated by a diffusion process with a continuous path. The key assumption in such an analysis is that incremental changes in the queue lengths are normally distributed. This leads to a characterization of the queueing network by a set of diffusion equations. The accuracy of the approximation depends on three factors: (i) approximation of a discrete-state process by a time-continuous Markov process, (ii) choice of proper reflecting barriers, and (iii) discretization of the continuous density function for queue lengths. Surprisingly, for the simple discrete Markov Chain model of Section 4, the diffusion approximation yields a result identical to that with exponential servers derived from Jackson's formulae. However, the main utility of the diffusion approximation in this context is that it can be used to analyze the effect of different coefficients of variation (ratio of standard deviation to the mean) for the service time distribution.

## 6. CONCLUDING REMARKS

Table 1 summarizes the characteristics of various models that have been discussed in this paper. Without a doubt the simplest model to use is the continuous time Markov chain model: the average number of busy Mp's, or the average number of busy Pc's, is simply  $n \cdot m / (n + m - 1)$ , where n is the number of Pc's and m is the number of Mp's. In many cases, however, it may be more realistic to model the memory cycle time as constant, rather than exponentially distributed,

and hence we developed the discrete Markov chain model in Section 4. Table 2 compares the continuous time and discrete time Markov chain models. In practice, it has proven useful to view these two models as bounds on the performance that will be achieved by the actual system; the continuous time Markov chain model is probably an overestimate of the variance of memory cycle time while the discrete Markov chain model is certainly an underestimate of the variance of the memory cycle time.

	Processing Time	Memory Cycle Time	Analysis	Computational Ease
Discrete Markov Chain	Constant $t_p = t_w$	Constant	Exact	Solution is algorithmic. Unwieldy for large $n$ .
Strecker's Approximation	Constant	Constant	Approximate	Closed form solution. Simple formula.
Continuous Time Markov Chain	Exponential	Exponential	Exact	Closed form solution. Simple formula.
Diffusion Approximation	Constant	Constant	Approximate	Closed form solution. Simple formula.
Simulation Model			Approximate	Unwieldy due to slow stochastic convergence.

1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
1.0000	1.5000	1.6667	1.7500	1.8000	1.8333	1.8571	1.8750
1.0000	1.6667	2.0476	2.2692	2.4095	2.5054	2.5748	2.6272
1.0000	1.7500	2.2701	2.6210	2.8630	3.0365	3.1657	3.2652
1.0000	1.8000	2.4102	2.8633	3.1996	3.4530	3.6482	3.8019
1.0000	1.8333	2.5059	3.0370	3.4533	3.7809	4.0415	4.2518
1.0000	1.8571	2.5751	3.1663	3.6486	4.0418	4.3636	4.6292
1.0000	1.8750	2.6274	3.2657	3.8024	4.2521	4.6294	4.9471
1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
1.0000	1.3333	1.5000	1.6000	1.6667	1.7143	1.7500	1.7778
1.0000	1.5000	1.8000	2.0000	2.1429	2.2500	2.3333	2.4000
1.0000	1.6000	2.0000	2.2857	2.5000	2.6667	2.8000	2.9091
1.0000	1.6667	2.1429	2.5000	2.7778	3.0000	3.1818	3.3333
1.0000	1.7143	2.2500	2.6667	3.0000	3.2727	3.5000	3.6923
1.0000	1.7500	2.3333	2.8000	3.1818	3.5000	3.7692	4.0000
1.0000	1.7778	2.4000	2.9091	3.3333	3.6923	4.0000	4.2667
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	11.1133	10.0018	8.5714	7.4056	6.4910	5.7671	5.1840
0.0000	10.0018	12.0922	11.8632	11.0645	10.1940	9.3794	8.6480
0.0000	8.5714	11.8982	12.7928	12.6790	12.1785	11.5519	10.9059
0.0000	7.4056	11.0904	12.6882	13.1829	13.1190	12.7844	12.3254
0.0000	6.4910	10.2119	12.1930	13.1266	13.4412	13.3985	13.1591
0.0000	5.7671	9.3899	11.5687	12.7939	13.4049	13.6218	13.5920
0.0000	5.1840	8.6549	10.9196	12.3369	13.1653	13.5957	13.7535

There are a couple of important considerations in the analysis of memory interference in multiprocessors that have not been touched on in this paper. The first is that many multiprocessors may not be bus bound, or  $t_p \neq t_w$ . For discussion of situations where  $t_p$  is greater or less than  $t_w$  see [1,13]. Another aspect in these models that needs to be examined more closely is the assumption that each processor accesses each memory module with equal probability. Program behavior, as well as the memory management policies of the operating system, may have a dramatic impact on these accessing probabilities. Measurement experiments are currently being designed for C.mmp to collect these processor to memory accessing frequencies.

#### REFERENCES

- Bhandarkar, D. P. Analytic Models for Memory Interference in Multiprocessor Computer Systems, Ph.D. Thesis, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa. (Sept. 1973).
- Beckenbach, E. (editor), Applied Combinatorial Mathematics, Wiley, New York, 1964.
- Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971.
- Feller, W., An Introduction to Probability Theory and its Applications, Vol. 2, Wiley, New York, 1966.
- Gaver, D. P., "Probability Models for Multiprogramming Computer Systems," JACM, Vol. 14, No. 3, July, 1967, pp. 623-638.
- Gordon, W. J. and G. F. Newell, "Closed Queueing Systems with Exponential Servers," Oper. Res., 15 (1967), pp. 254-265.
- Jackson, J. R., "Jobshop-like Queueing Systems," Management Sci., 10, 1 (Oct. 1963), pp. 131-142.
- Kobayashi, H., "Application of the Diffusion Approximation to Queueing Networks: Part I - Equilibrium Queue Distributions," 1st Annual SIGME Conference on Measurement and Evaluation, March, 1973, pp. 54-60.
- McCredie, J. W., "Analytic Models as Aids for Multiprocessor Design," Proc. of the 7th Annual Princeton Conference on Information Science and Systems, March, 1973.
- McKinney, J. M., "A Survey of Analytic Time Sharing Models," Computing Surveys, Vol. 1, No. 2, pp. 105-116, 1969.
- Newell, G. F., Applications of Queueing Theory, London, Chapman and Hall, 1971.
- Skinner, C. and J. Asher, "Effect of Storage Contention on System Performance," IBM Sys. J., Vol. 8, No. 4, 1969, pp. 319-333.
- Strecker, W. D., Analysis of the Instruction Execution Rate in Certain Computer Structures, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1970.
- Wulf, W. A. and C. G. Bell, "C.mmp - A Multi-Mini-processor," AFIPS FJCC Proc., 1972, Vol. 41, Part II, pp. 765-777.

# INTERCONNECTING A DISTRIBUTED PROCESSOR SYSTEM FOR AVIONICS

George A. Anderson  
Senior Research Engineer  
Systems and Research Center  
Honeywell Inc.  
Minneapolis, Minnesota

## ABSTRACT

This paper describes the interconnection scheme devised for an advanced Air Force system concept called Distribution Processor/Memory (DP/M) in which topologically irregular networks of small computers are used to perform avionics processing. The interconnection scheme involves the use of a combination of global and point-to-point busses to handle message traffic in predominantly homogeneous systems of from 5 to 20 computers. The major features of the scheme are the use of biphasic bit-serial transmission, associatively addressed messages, and a method for reconfiguration of the point-to-point communications paths under program control. It is expected that the scheme may have general applicability to other distributed processing systems, particularly other real-time systems employing limited-capability processors.

## INTRODUCTION

The problems involved in interconnecting a multi-computer system, particularly when "multi" means three or more, are well known. Tradeoffs in the design involve factors such as the cost of busses versus their speed, their complexity versus their load on the computational resources of the system, their reliability and its effect on system reliability, ad infinitum. This paper presents a particular interconnection scheme\* developed to fit a specialized environment, but one which may have more general applicability in computer networks. This scheme involves the interconnection of processors by a single global bus together with a nonregular network of processor-to-processor links. These links are switchable to allow configuration of a variety of data paths during operation. The resulting paths are used both as a primary communications medium and as a backup for the global bus. An associatively addressed message transmission scheme for transfers on both the busses provides for intercommunications with little degradation of computational capability, even for large (over 20 processor) systems.

## PROBLEM BACKGROUND

The DP/M concept is essentially the use of a varying number of simple and identical processor/memory elements (PEs) to handle a wide range of avionics system-processing requirements. System sizes are expected to range from five to seven PEs on undemanding missions to over 20 PEs in complex environments. Each PE represents memory of 4K words and computation rate of about 250 thousand instructions per second (KIPS) on avionics problems, so this means system capacities will range from 1000 to 5000 KIPS. It is the job of the interconnection scheme to allow this level of modularity and the variability in system size by providing efficient communications between the components of the system without itself

becoming an undue consumer of processing resources, a reliability handicap, or a costly resource in itself.

The DP/M avionics processing load is partitioned into a number of relatively autonomous functions which communicate primarily via an "aircraft state vector" of a few hundred bits. These functions are further broken down into subfunctions with well-defined boundaries and low intercommunications requirements. An example of a major function is flight control, which may be separated by axis and by axis subfunctions into at least six units, called processes. As a test case during DP/M concept development, a very demanding environment was hypothesized and broken down into approximately 50 individual processes. Each process in the decomposition is of low complexity, with typical requirements of under 150 KIPS and 2K memory words.

In such a decomposition, communication within the system is of two distinct types--interfunctional and intrafunctional. Including Exec overhead, the former is estimated at under 200 thousand bits per second, while the latter may be up to 300 Kbits per second. The interfunctional transfers are typically short messages such as Exec commands and state vector information, while the intrafunctional transfers tend to be longer, consisting of data block moves. Interfunctional transfers involve all processors at one time or another, while intrafunctional transfers are localized to the few processors in which the function is performed.

Physical constraints on the interconnection scheme were quite limiting. From the beginning, it was determined that the system would be physically distributable around the aircraft and that the interconnection scheme should thus allow this distribution with low cost. Also, the software goal was to have maximum commonality between systems of different sizes, so the interconnection could not change character as system size varied. Finally, since the interconnection is the major "central" system resource, it had to be amenable to fault tolerance techniques and provide a low-cost redundancy option. A simplifying assumption was that the system I/O to sensors, actuators, etc., would be handled directly by the processors and not through the interPE connections.

## DESIGN APPROACH

The design of the interconnection scheme proceeded simultaneously with the definition of the processing elements, the software and the requirements analysis. As such, it had ample time for iteration and consideration. Integrated bussing/processing approaches like the Holland machine (2) and the distributed processor of Burnett and Kozcela (3, 4) were rejected early in the work because of software problems, leaving the bussing work to proceed almost independently of the PE definition. The approaches used by a number of

\*The scheme is a result of work done by the Honeywell Systems and Research Division for Wright-Patterson Air Force Base in the development of a Distributed Processor/Memory (DP/M) system to serve general avionics processing needs in the late 1970s and early 1980s (1).

advanced architectures like the Navy AADC (5) and the Burroughs D-machine (6) were considered. These were uniformly rejected, however, when the system bandwidth requirements became known. It was found that, up until the present, design approaches had largely been devoted to high-rate intercommunication between computers via memory modules, either by multiprocessing like the D-machine, or by partial sharing of memory like the CDC 6500 and others. [An exception to this is IBM's ASP configuration for two computers (7).] The unique characteristics of the avionics environment, however, obviated the need for massive amounts of shared data and, indeed, argued against shared memory approaches for fault tolerance reasons (protection of data).

Another characteristic of the more general-purpose approaches was their regularity. In order to handle a variety of processing loads, these machines had provided very regular interconnection schemes in which the access rights of a processor to other processors or to memory were largely independent of its location in the system. The Solomon (8) architecture is a good example of this. In contrast, the DP/M environment involved a known and nonregular pattern of intercommunications between processes and a general level of global (interfunctional) communications. Furthermore, except under unusual conditions such as reconfiguration to mask failures, the association of processes to processors was static, so interprocessor communications could be considered irregular and quasistatic. These differences, combined with the low data rates, the requirement for physical distribution, and the requirement for fault tolerance, indicated that a new approach to computer interconnection might best solve the specific problem to which DP/M was addressed.

The bussing scheme chosen, shown in Illustration 1, is a hybrid, combining a global bus visiting each PE with a number of point-to-point busses between PEs in an irregular pattern. Both busses are bit-serial, biphasic coded, with data transfer rates of 1 Mbit. The global bus is provided for the interfunctional data transfers and the local busses for the intrafunctional transfers and as a backup to the global bus. A distinctive feature of the scheme is that the local busses are switchable; each PE includes hardware by which, under program control, the busses attached to it may be connected to each other, to the PE itself, or may be idle. Illustration 2 shows examples of the use of this capability. A possible physical interconnection is shown in 2a. Here, the maximum number of busses to any PE (exclusive of the global connection) is three. A combination of switch settings which configure a quasiglobal bus is shown in Illustration 2b. This is an example of what might occur during recovery from a failed global bus. In Illustration 2c, a combination of switch settings is shown which configures two so-

ILLUSTRATION 1  
DP/M Hybrid Bussing

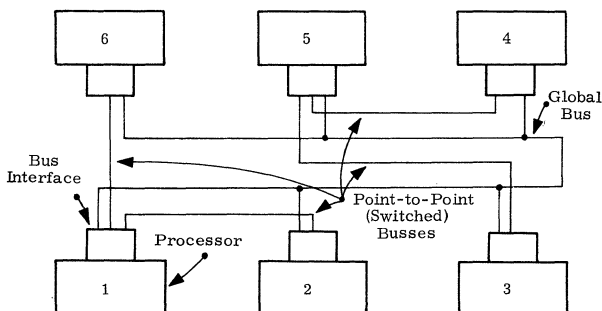
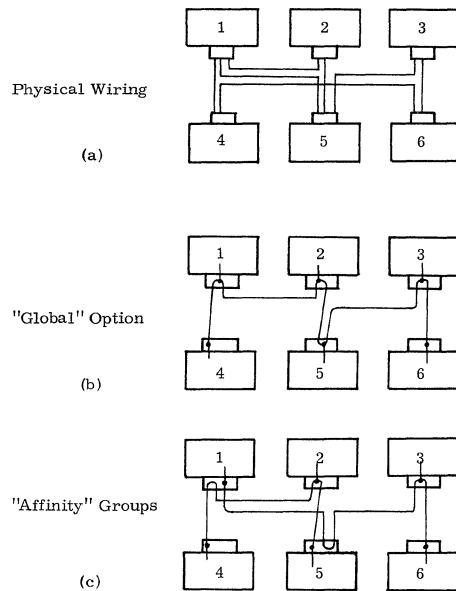


ILLUSTRATION 2  
Switchable Bussing Alternatives



called "affinity groups" of PEs which may communicate independently of the global bus for intrafunctional transfers.

The hybrid approach provides a distinct advantage over a single, possibly faster, global bus. First, as system requirements grow and change, the option of nonregular point-to-point interconnection is expected to allow more cost-effective expansion by requiring only useful interconnections. Secondly, the extra bandwidth can be concentrated in physically localized areas of the system instead of requiring overall high bandwidth and, in fact, may result in a very high total data transfer rate achieved by simultaneous use of many slow paths. Finally, the two-type approach can be used to provide redundancy for fault tolerance as, and when, needed rather than on an all-or-nothing basis.

Provision of switchability in the local busses is primarily for fault recovery and flexibility reasons. In case of a processor or local bus failure, relocation of processes may be required, negating the effectiveness of a dedicated approach to interconnection. Using the switchability, however, an alternative switch pattern can be used to provide the same intercommunication paths to the now relocated processes. Also, in case of a global failure, a quasiglobal bus can be configured to handle some or all of the previous global bus traffic. In this case, too, the system designer can choose to spare the global bus with a complete set of switchable busses or he can use some or all of the connections primarily intended for intrafunction traffic. As will be shown below, the bus hardware supports such reconfiguration to the extent that the reconfigured interconnection may be totally invisible to the software.

## DETAILED DESIGN

### ADDRESSING MECHANISM

In order to minimize the overhead involved in process relocation within the DP/M system, as well as to make the geometry of the system and of the interconnection scheme invisible to the software, it was determined early in the design that physical addressing

of messages on the communications system was undesirable. Software that was transferred between systems of various sizes as well as software operating before and after process relocation could not be easily provided with enough information to physically address its messages. In a system like DP/M, tables for such addressing would have been difficult, if not impossible, to maintain during mission phase changes and reconfiguration after failure. As an alternative to physical addressing, it was decided to place in each PE's bus interface enough hardware to support associative addressing of messages and to require each transmission on a bus to be preceded by a destination "name." Each process in a PE is required to place in the appropriate interface registers a "name" by which it was known in the system. The bus interface then, has the responsibility of handling a list of these names in associative memory fashion, matching message traffic on the bus against names and accepting messages destined for processes within the PE.

It was determined further that the names of processes tended to be hierarchial in nature; that is, a process might be identified as: "Flight Control, Y Axis, Stability Augmentation Loop," and that it was desirable to allow messages to be directed either to a particular component process by using its full identification or to other levels of the naming "tree." To accomplish this without requiring each process to specify multiple names, destination names transmitted by processes were made of variable length and the associative matching performed by the interface is on a bit-by-bit basis. Thus, the name specified by the process wishing to receive messages is its full identification, but it is given all messages whose specified destination matches the name in all transmitted bits. Note that this type of scheme allows both one-to-one and one-to-many type transmissions.

#### TRANSMISSION SCHEME

Although electrical design of the bus has not begun, preliminary work and the results of other work (9, 10) indicate that a biphas coding scheme is optimal for the low data rates and physical environment foreseen for DP/M. The message format on the busses, using a biphas coding, is shown in Illustration 3. The first bits of the message contain the destination name interspersed between 1s at even bit times. Following the first zero at an even bit time, the remainder of the transmission is message content, with no further "tag" bits. In this way, the variable-length name is uniquely

delimited with minimum wasted bandwidth. To simplify the hardware, names are restricted to be less than or equal to the PE's word size, currently either 16 or 24 bits. Following the name, the data transmission is to be an integral number of words. To be compatible with a proposed Air Force multiplexing standard (10), the bus clock rate will be 2 MHz, yielding a 1M bit raw transfer rate.

Busses are allocated on a round robin basis, with each PE on a bus being provided with opportunity to transmit or "pass" in turn. Control passes from one PE to another when a PE in control transmits a biphas synch pulse (a pulse more than one bit-time in duration). Each PE has two registers in its bus interface, a Bus Length register and a Position register indicating its position on the bus. Whenever a synch pulse is transmitted on the bus, every PE increments a Current Control counter containing the bus position number of the PE which currently has control of the bus. In one PE, this number matches the Position register. This PE is in control of the bus, and has the option of transmitting a message or passing control. To transmit a message, the PE simply begins emitting the biphas code as shown in Illustration 3, terminating the transmission (and its control of the bus) with a synch pulse. If it has no transmission ready, it simply emits a synch pulse, causing control to pass on. Thus the minimum time between transmissions from a PE is the time it takes for control to cycle around when every other PE on the bus emits only a synch pulse. This latency time is expected to be under 5 microseconds per PE, but is highly dependent on final electrical design, physical separation of PEs, etc. After overhead for allocation and destination header transmission, the busses are expected to provide information transfer rates in excess of 500 Kbits, a safety factor of more than 2:1 over anticipated requirements.

#### BUS SWITCH DESIGN

As part of the study work, a preliminary design for the bus switch and interfaces was performed. A block diagram of the switch is shown in Illustration 4. The PE interfaces to the global bus and to a number of local busses via receiver/drivers which resistively couple to a balanced pair. The tee in the global bus is presumed to be external to the PE, while local busses are expected to connect to only two PEs. Inside the switch, the busses are separated into receive, transmit, and transmit key signals, which then fan to a number of crosspoint switches, shown in the detail. The contents of switch control registers control the crosspoints to effect the switching as shown in Tables 1, 2 and 3.

The PE is provided with two blocks of essentially identical interface hardware, one for the global bus and one which may be switched onto any of the local busses. In normal operation, the crosspoint shared by the global bus and the global bus interface is closed, while other crosspoints are closed as required. Alternative crosspoints are provided, however, to allow reconfiguration such as in Illustration 2a. Note that by reconfiguring in this way, the software continues to use the global communications facility in exactly the same way, with only the bits in the switch control registers and possibly the bus control registers (in the bus interface) being altered.

As can be seen from the tables, all combinations of two and three local busses can be interconnected via the crosspoints and buffers. As an example, to connect local bus A to local bus B, crosspoints one and two are closed, connecting A and B via a buffer. If, in addition, the PE itself is to be attached to the bus thus configured, crosspoint 3 is also closed.

ILLUSTRATION 3  
Example Message Format

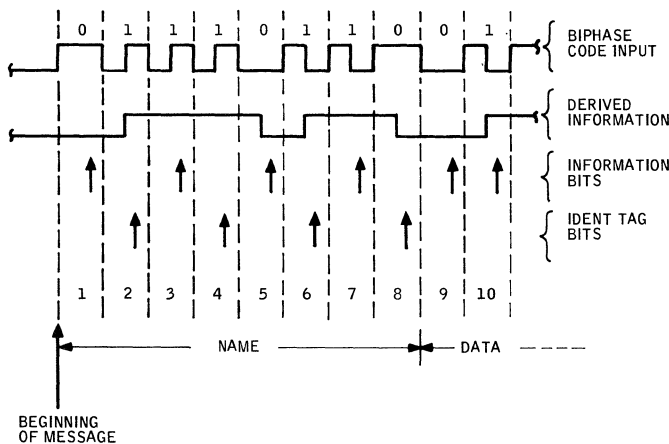


ILLUSTRATION 4  
Bus Switch

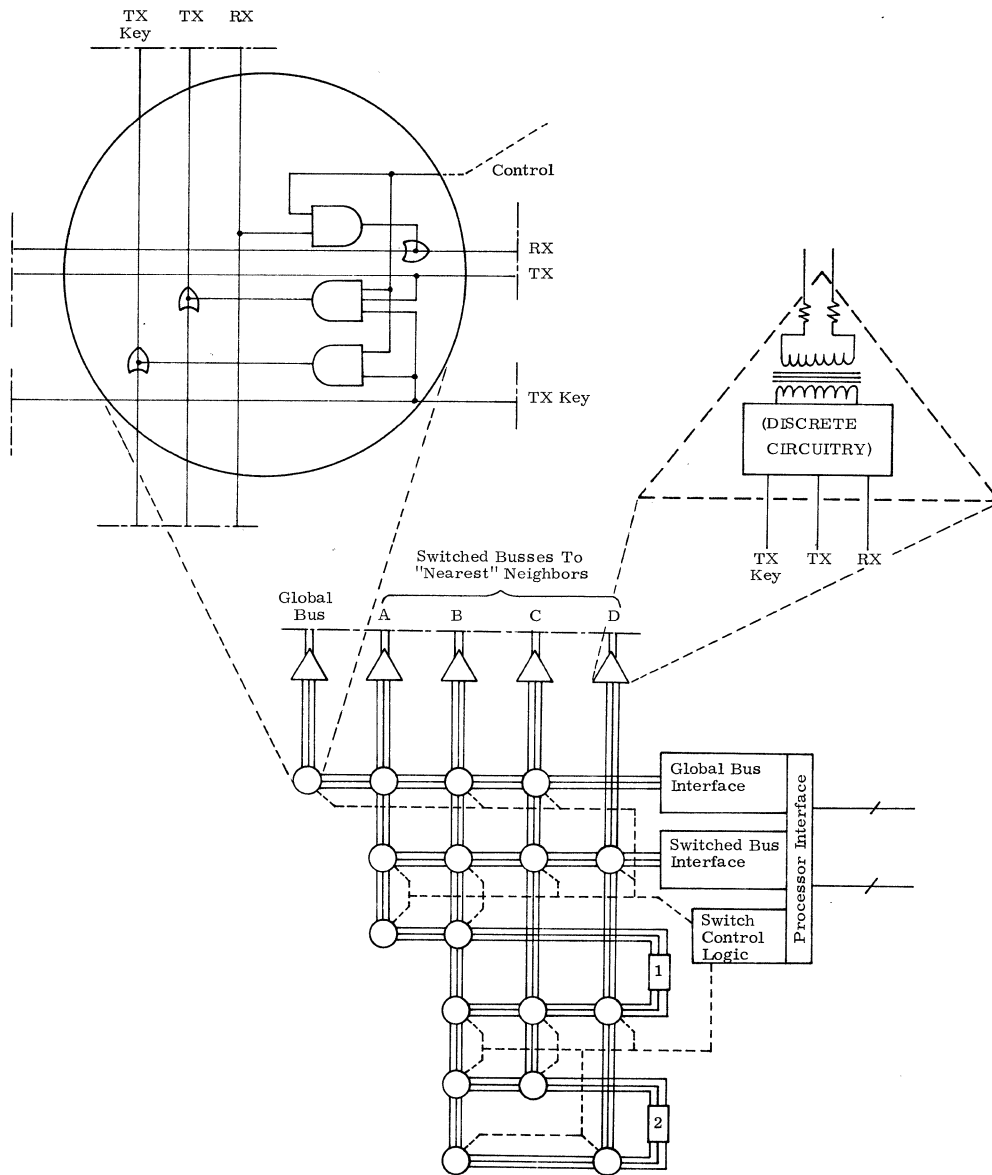


TABLE 1  
Global Bus Interface Connection Codes

Octal	Connection
0	Global Bus
1	Switched Bus A
2	Switched Bus B
3	Switched Bus C

TABLE 2  
Switched Bus Interface Connection Codes

Octal	Connection
0	Switched Bus A
1	Switched Bus B
2	Switched Bus C
3	Switched Bus D

TABLE 3  
Switched Bus Buffer Connection Codes

Octal	Buffer 1		Buffer 2	
	S1	S2	S1	S2
00	A	-	B	D
10	A	B	B	D
20	A	C	B	D
30	A	D	B	D
01	A	-	B	C
11	A	B	B	C
21	A	C	B	C
31	A	D	B	C
02	B	C	C	D
12	A	B	C	D
22	A	C	C	D
32	A	D	C	D
03	A	-	C	D
13	A	B	-	-
23	A	C	-	-
33	A	D	-	-



It is obvious that the existence of a bidirectional buffer circuit is essential to the success of the scheme. Several TTL designs of such a circuit have been performed and a small breadboard has been constructed. As design of the system progresses and more information on clocking techniques, etc., is available, a full-scale breadboard consisting of several switches and interconnecting busses is planned.

#### PE INTERFACE HARDWARE

Block diagrams of the interfaces between the processor portion of the PE and the busses are shown in Illustrations 5 and 6. Both are provided with identical encoding/decoding hardware and name recognition circuitry. It should be noted that the provision of four name registers and four local bus connection points was somewhat arbitrary. The requirements for both are expected to be determined more definitively for DP/M in later work.

On the output side, both interfaces provide channel-type hardware which allows the software to simply specify the memory location of the message to be transmitted, the length of the transmission, and the number of bits in the destination name. The channel then gains control of the bus, transmits the requisite header using bits from the first memory word, then transmits the remaining memory words as data.

On the input side, the global interface is provided with queuing storage to allow incoming messages to be accepted by the processor in FIFO order. Interrupts are provided to indicate receipt of a complete message and impending queue overflow. In order for the software to determine the destination of the message, the destination address as received on the bus is included at the beginning of the queued information.

For the local bus interface, where longer transmissions are expected, an input channel is provided to place the arriving information in a software-specified main memory buffer area. The input channel supports automatic double buffering of arriving information, allowing the processor a great deal of time before data is overwritten. Here, as well as in the other interface blocks, various status flags and interrupts have been provided to the processor.

Fault detection in this preliminary design is performed in two ways. First, a PE which misses its control slot on the bus may effectively block all further use of the bus by not propagating its synch signal. This is detected by a timer in each PE's allocation logic which interrupts the processor, indicating "bus assign failure" if an excessively long period of silence is observed on the bus. Secondly, missed bits during data transfers on the bus are detected on the incoming side by maintaining a modulo word-length count of the arriving data. If, at the end of the transmission, this count is nonzero, the processor is interrupted. This detects any missing bits in the data portion of a transmission only. Errors in the destination header portion of the transmission are expected to require software detection, since it is likely that errors will cause the message to be either missed by all PEs or to be accepted by the wrong one or ones.

#### CONCLUSIONS

Although the result of preliminary work, this interconnection scheme offers several unique features which may be of general interest. Among them are:

- 1) Use of variable-length, associative addressing of inter-PE messages. As systems grow in complexity by distributing

the computing function, this may become a cost-effective way of relieving the software of the addressing burden. It is quite analogous to the use of symbolic rather than absolute addresses in assembler and HOL programming with the extension to dynamic mapping of addresses onto hardware.

- 2) Provision of flexibility and fault-tolerance through the use of switchable interconnections between processors. This technique provides a decentralized switching system which can be dynamically adapted to the needs of a particular problem phase.
- 3) The use of relatively complex hardware to reduce significantly the communications and control overhead conventionally found in multiprocessor and multicomputer systems.

Much validation work remains to be done on this preliminary design, but much of interest has already been accomplished. Work of an architectural nature remains to be done to assess the general applicability of this type of computer system, particularly for more demanding problems. In situations where functions do not easily decompose with low intercommunications, extensions of the concept to higher bandwidth busses may be considered. Theoretical investigation into the physical and virtual interconnection networks possible from switchable busses may be of great value. It already appears that heuristic approaches to determining interconnection patterns and switch settings may be difficult to develop. Certainly, the concept of an adaptive system with some intelligence, rather than just a system which chooses from interconnection templates, is worth investigating. As low-cost mini- and micro-computers become available, the potential for cost-effective distributed systems appears to be increasing, and with it the interconnection problems of such systems.

#### REFERENCES

1. Johnson, M.D., et al. All Semiconductor Distributed Aerospace Processor/Memory Study. Final Report, Volume 2, Air Force Avionics Laboratory, Wright-Patterson AFB, Ohio. AFAL TR-73-226
2. Holland, John. "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously." Proc. EJCC, pp. 108-113, 1959.
3. Koczela, L.J. Study of Spaceborne Multiprocessing. Final Report - Phase 1, Volume 2, 15 April 1967, National Aeronautics and Space Administration Electronics Research Center, No. C6-1476.10/33.
4. Burnett, G.J., et al. "A Distributed Processing System for General-Purpose Computing." Proc. FJCC, pp. 757-768, 1967.
5. Thruher, K.J., et al. Master Executive Control for the Advanced Avionic Digital Computer. Interim Report Volume 1: Summary, Honeywell No. Z9506-3018, June 1972.
6. Davis, R.L., et al. "A Building Block Approach to Multiprocessing." Proc. SJCC, pp. 343-349, 1970.
7. Lorin, H. Parallelism in Hardware and Software: Real and Apparent Concurrency. Prentice-Hall, pp. 166-176, 1972.
8. Slotnik, D., et al. "The Solomon Computer," Proc. FJCC, pp. 97-107, 1962.
9. Barnes, B.P., et al. Application of Information Transfer Techniques for Solving the Internal Communication Requirements of an Advanced Manned Bomber. AFAL TR-72-209.
10. Proposed Standard for Aircraft Multiplex Data Bus, Air Force Avionics Laboratory, Wright-Patterson AFB, Ohio, 2 March 1973.

ILLUSTRATION 5  
Bus Interface (Switched Bus)

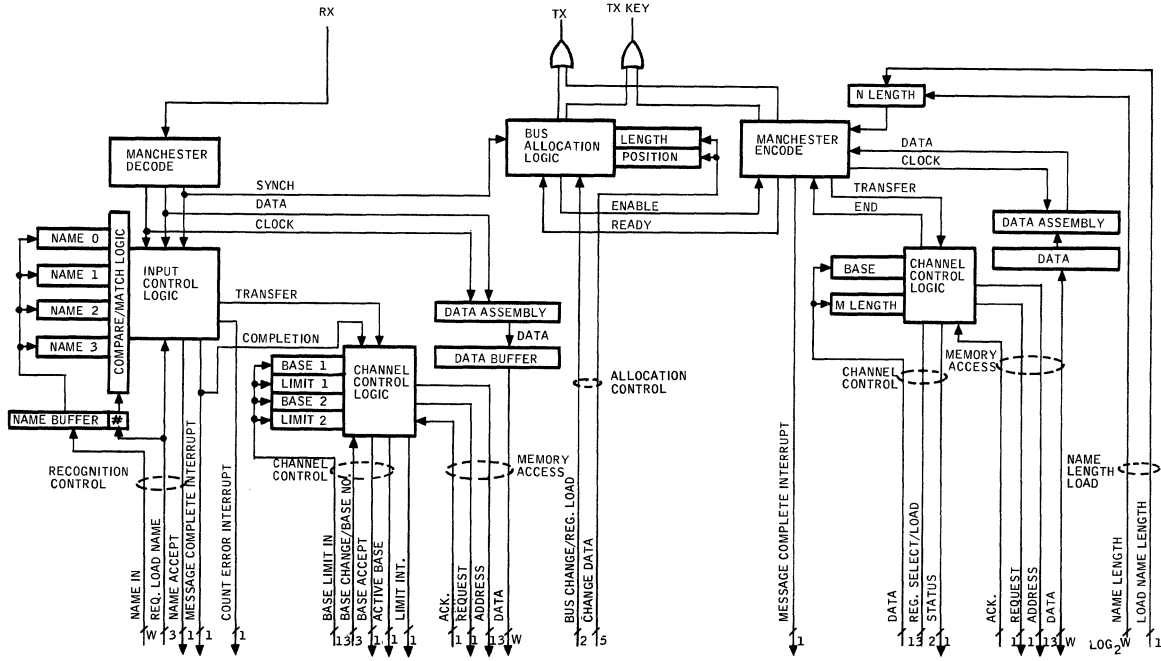
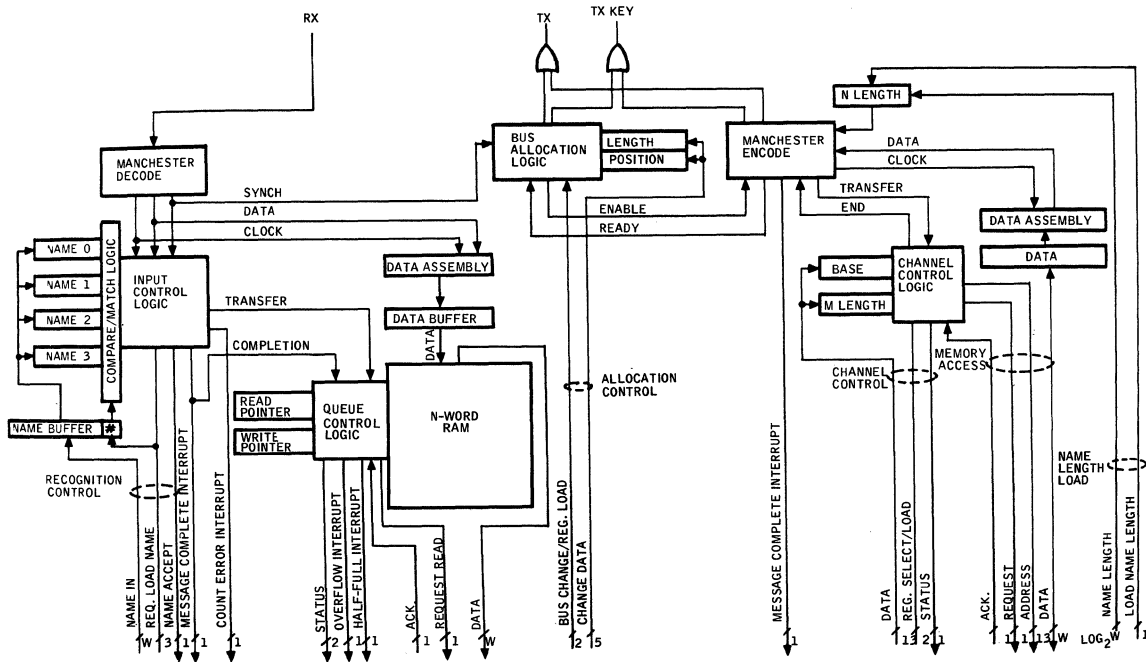


ILLUSTRATION 6  
Bus Interface (Global Bus)



# BANYAN NETWORKS FOR PARTITIONING MULTIPROCESSOR SYSTEMS

L. Rodney Goke  
*Texas Instruments*  
*Austin, Texas*  
 and  
 G. J. Lipovski  
*University of Florida*  
*Gainesville, Florida*

## 1. INTRODUCTION

Restructurable computing systems using multiple miniprocessors are currently of interest and promise advantages over large single processor time-shared systems for some applications (1-3). The modular nature of such systems can offer graceful degradation, improved availability, and expandability. Such systems to date have generally contained a small number of processors and have used one or more switching structures based on a crossbar.

It is now reasonable to expect that the low cost and high cost/performance of mass produced LSI microprocessors will make systems with much larger numbers of processors practical (4). Modules of other resources, such as memory and I/O, might also be more numerous in such systems.

The number of contacts, or switching devices, for a crossbar, however, increases with the square of the number of connections to it, making it prohibitively expensive for very large systems. Since the fanout of switching devices in a crossbar increases linearly with the connections to the structure, this too can be a problem in large systems, especially when expandability is not to be limited. It is thus increasingly desirable to find structures better suited than the crossbar to partitioning large systems.

This paper describes a class of partitioning networks, called banyans, whose cost function grows more slowly than that of the crossbar and whose fanout requirements are independent of network size. Such networks can economically partition the resources of large modular systems into a wide variety of subsystems. Any possible partition can be realized by paralleling several networks or by multiplexing a single network in a manner to be described later. Results will be given indicating that a cost/performance advantage over the crossbar can be obtained for large systems and that the crossbar can, in fact, be considered a non-optimal special case of a banyan network. Inherent fail-soft capability and the existence of rapid control algorithms which can be largely performed by distributed logic within the network are also important attributes of banyans.

This paper presents fundamental properties and preliminary simulation results of banyan partitioning networks. A more detailed treatment, including proofs of theoretical properties, is reserved for reference (5).

## 2. PARTITIONING

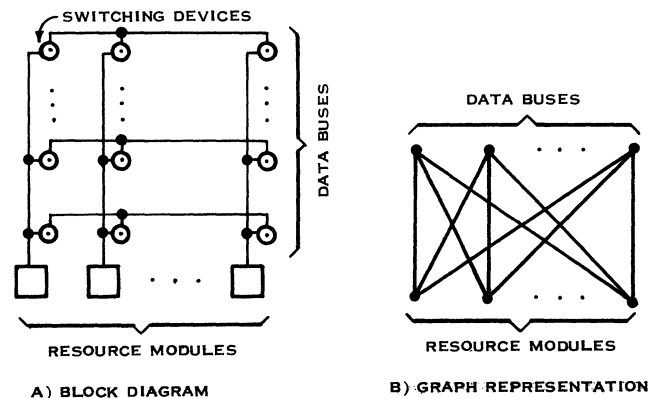
The purpose of a partitioning network, as considered here, is to partition the resource modules of a system into disjoint subsystems by effectively providing a separate bidirectional data path connecting the resources in each subsystem. Once connected, the resources of a subsystem could communicate by time sharing this data path in a manner similar to that used in such systems as the PDP-11 (6) and the HP 3000 (7).

### 2.1 CROSSBAR NETWORKS

The crossbar network shown in figure 1a is perhaps the most straightforward partitioning structure. For  $N$  resource modules, it contains  $[N/2]$  data busses the maximum number of nontrivial subsystems possible at one time. A subsystem with only one resource is trivial because it does not need the structure to communicate with itself. This network requires  $N[N/2]$  bidirectional SPST switching devices, <sup>1</sup> each of which is connected to  $N-1$  identical devices by a data bus.

Figure 1b is a graph representing the same structure. This representation is similar to that used by Benes (9) and uses vertices to represent data busses or links, and edges to represent the switches con-

Figure 1. Crossbar Partitioning Network



<sup>1</sup>Bidirectional electronic switching devices suitable for all networks in this paper are discussed in reference (8).

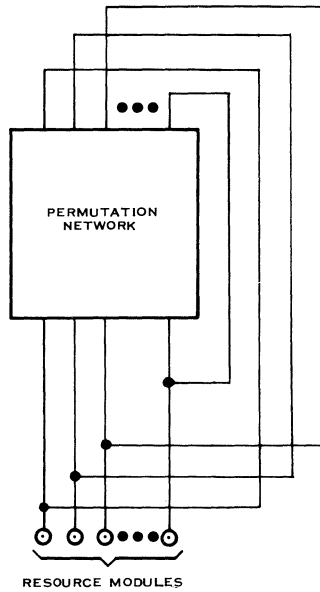
necting them. Note that the crossbar is represented by a biparte graph with an edge connecting each bus with every resource module. Graph representations will be used with other structures later.

More specialized crossbar structures have been used in a variety of multiprocessor systems (3, 10-12).

## 2.2 PERMUTATION NETWORKS

It is possible to build a partitioning network from a permutation network by supplying the external links shown in figure 2. A permutation network can connect, in pairs, a set of input terminals to a set of output terminals of equal size so that any desired permutation of inputs onto outputs can be realized. These connections allow transmission in either direction when bidirectional switches are used in the network. In the configuration of figure 2, the network permutes the set of resource modules onto itself, allowing connected subsystems to correspond to the cycles of the permutation. By choosing a permutation with the appropriate cycles, any desired partition can be connected.

Figure 2. Permutation Network used as a Partitioning Network



This result is theoretically significant because it implies that an N-terminal partitioning network does not need to contain any more contacts than an N-input, N-output permutation network. It has been shown that when N is a power of 2, such a permutation network can be built with as few as  $4(N \log_2 N - N + 1)$  contacts (13-15).

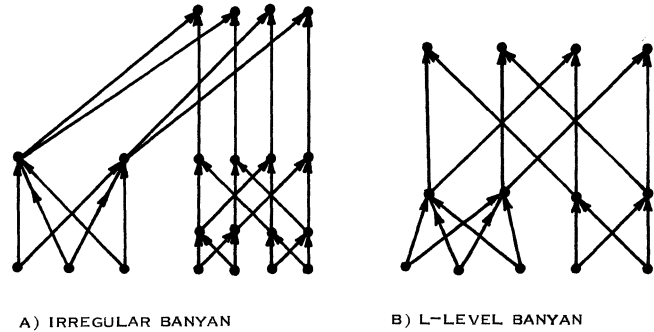
The partitioning structure of figure 2 is of limited practical value, however, because of excessive propagation delay in large subsystems. A signal in the data path connecting a subsystem with i resource modules may have to propagate through the permutation network as many as  $\lceil i/2 \rceil$  times to reach its destination. Each time through, it must propagate through as many as  $(\log_2 N - 1)$  contacts. Control of this structure would also be relatively complex and could limit restructuring speed.

## 3. BANYANS

A banyan network, named for the East Indian fig tree of somewhat similar structure, is defined in terms of its graph representation. The graph of a banyan is a Hasse diagram of a partial ordering (16) in which there is one and only one path from any base to any apex. A base is defined as any vertex having no arcs incident into it, an apex is any vertex with no arcs incident out from it, and all other vertices are called intermediates. When used as a partitioning

network, the bases are connected to resource modules, while the apexes and intermediates are within the network. Some examples of banyans are shown in figure 3. We use a directed graph representation because it is useful for specifying the structure and its control algorithms, but the switches represented by the edges are still bidirectional.

Figure 3. Examples of Banyans



### 3.1 TREE-SHAPED CONNECTIONS IN A BANYAN

In a banyan the data path established to connect the resource modules of any subsystem always forms a tree rooted at some apex. By definition there is a unique path from each base to each apex. A subsystem is connected by selecting an apex and then closing all switches along the path from each desired base to the selected apex. Since each path is unique, the resulting data path forms a tree rooted at the apex. Algorithms for locating eligible apexes and establishing the connections will be presented in section 3.3.

Tree-shaped data paths are significant because they can afford low propagation delay with limited fanout and because they lend themselves well to the inclusion of priority hardware (17). Propagation delay and fanout will be discussed later. Priority hardware is desirable in any data path used as a time-shared bus in order to resolve conflicts when two or more resources request bus control simultaneously. Details of how priority hardware can be built into a banyan network can be found in reference (5).

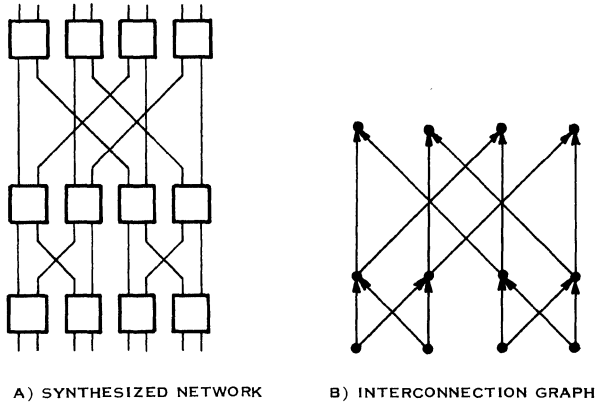
### 3.2 SYNTHESIZING LARGE BANYANS FROM SMALL ONES

Large banyan networks can be synthesized recursively from smaller ones. Suppose that one has available a number of small banyan networks, perhaps supplied by a manufacturer as a basic module, and one wishes to synthesize a larger network. This can be done as illustrated in figure 4a by connecting the apexes of some banyans to the bases of others.

The interconnections of these banyans can be represented by a graph, as illustrated in figure 4b. In this graph, each vertex represents a banyan network. An arc from any vertex V1 to another vertex V2 means that one apex of banyan V1 is directly connected to one base of banyan V2. We assume that if there are any arcs incident into a vertex, then the corresponding banyan has exactly one base for each incident arc.

Similarly, the number of apexes equals the number of arcs incident out from the corresponding vertex unless there are none. When there are no arcs incident into a vertex, the bases of the corresponding banyan become the bases of the synthesized network. Similarly, the apexes of the synthesized network are those of the component banyans with no arcs incident out.

Figure 4. Banyan Synthesis



**Theorem 1:** When banyan networks are interconnected as described above, the resulting network will be a banyan iff the graph of the interconnections is a banyan graph.

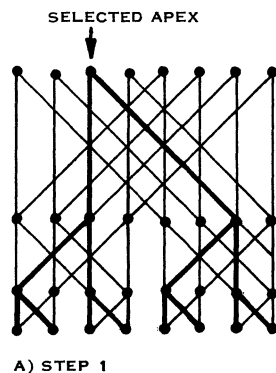
**Proof Sketch:** There are three ways that a directed graph can not be a banyan; one, if it contains a circuit, two, if there is more than one path from some base to some apex, or three, if there is no path from some base to some apex. Since the component networks are banyans, any of these conditions in the interconnection graph would cause the same condition to exist in the graph of the synthesized network, and vice versa.

This theorem is important because once one or more banyan structures are known, these structures can be recursively expanded to arbitrarily large sizes. The SW structure, discussed later, is based on recursive expansion of the crossbar, one of the simplest banyan structures.

### 3.3 CONTROL OF CONNECTIONS

Figure 5 illustrates how a data path connecting an arbitrarily selected apex with any desired subset of bases can be established in two steps. Set-up is facilitated by a single control line provided in each link of the network. First, a "one" signal is broadcast baseward from the selected apex over the control line, as illustrated in figure 5a. The signal fans baseward at each vertex so that the "one" propagates to all bases. This signal sets a flip-flop in each intermediate and apex through which

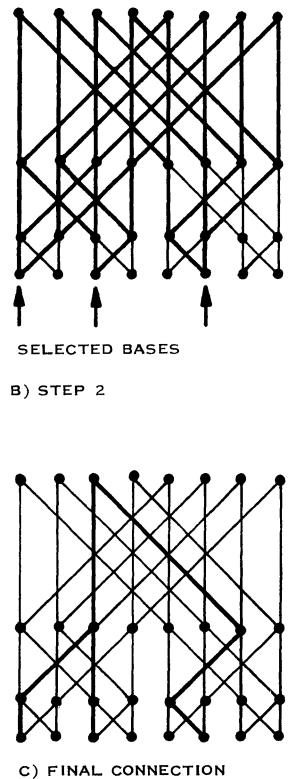
Figure 5. Set-up Algorithm



it passes.

In the second step, "ones" are broadcast apexward from each base in the desired subsystem, as illustrated in figure 5b. In this step, the signal is OR'ed apexward at each vertex. As illustrated in figure 5c, the desired connection is made by closing every switch that receives this signal from below and has a set flip-flop in the adjacent vertex above. These are the links through which control signals propagated in steps one and two.

Figure 5. (Cont.)

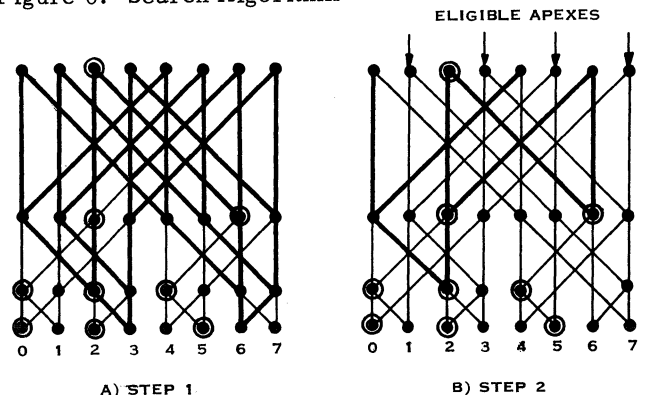


As described, this set-up algorithm would require two steps but only one control line in each link. Unlike the data lines, this control line is always connected between vertices and does not require a bi-directional switch for each edge of the graph. Switching for the control line occurs at the vertices where the signal is either OR'ed up or OR'ed down.

Any apex may be used in connecting the first subsystem, but subsequent apexes must be selected so that the new connection does not overlap with any vertex already in use. A two-step search algorithm for identifying the eligible apexes is illustrated in figure 6. In this example, the circled vertices represent those already in use, and bases 3 and 6 are to be connected as a new subsystem. As shown in figure 6a, control signals are first broadcast apexward simultaneously from all bases in the desired subsystem and are then OR'ed upward using the same control line used in set-up. During this step, a flip-flop is set in every intermediate and apex which receives this control signal and is already in use.

In the second step, illustrated in figure 6b, the control signals from the bases are turned off, and each

Figure 6. Search Algorithm



vertex with a set flip-flop broadcasts a "one", which is OR'ed apexward on the same line used in step one. All apexes not receiving a "one" during this step are eligible. Final selection could then be performed by a priority circuit attached to the apexes.

Steps one and two of this algorithm, like those of the set-up algorithm, could be combined using a second control line. With four control lines, search and set-up could all be combined in one step.

In the event of a hardware failure, any vertex could be effectively removed from the network by disconnecting all data lines to it and by treating it as if it were always in use. New connections would then be routed around the faulty cell <sup>2</sup>.

### 3.4 PARALLEL AND MULTIPLEXED NETWORKS

In partitioning a system, the search and set-up algorithms are repeated until all subsystems have been connected or until no eligible apex can be found. Although most subsystems might be connected this way in practice, a banyan may not always be able to connect all subsystems of a partition simultaneously. When subsystems are associated with independent jobs, this would imply only that the partitioning network be a limited resource for which jobs must compete much as they do for other system resources. When a subsystem cannot be connected under existing conditions, the associated job could be held in a queue until enough other subsystems were dissolved to permit the connection.

If, however, one wishes to simultaneously connect more subsystems than can be accommodated with a single banyan, there are two solutions. <sup>3</sup> First, several banyans can be connected in parallel. The parallel networks would function independently but their bases would be connected to the same set of resource modules. As many subsystems as possible would be connected in the first network. Those left over would be connected in as many additional networks as required.

The other solution is to multiplex a single network so that it periodically rearranges itself to connect first one set of subsystems, then another, and so on, so that each subsystem has some time slot during which it can communicate. A partitioning network, as considered here, acts as a rearrangeable set of time-shared buses. A resource module attached to the network must request and receive control of its bus before transmitting data, and must be prepared to wait whenever the bus is not immediately available. Normally the bus would be unavailable only when used by other resources in the same subsystem; but should it ever become temporarily unavailable for other reasons, the only effect would be to delay data transmission within the subsystem. This situation makes multiplexing pos-

sible with little or no modification of the resource modules. The system need only be designed so that any resource not currently connected by the network would "see" it as a busy bus.

Multiplexing requires that a small amount of memory be associated with each switch in the network to store the state of the switch during each time slot. With LSI this could be done at reasonable cost by associating a small register with each switch and synchronizing all state changes from a central clock.

The techniques of parallel networks and multiplexing may be mixed to balance cost and performance. Whether a network structure is space shared with parallel hardware or time shared with multiplexing, the parallel networks and/or time slots share many properties and are called layers. The number of layers required depends on a number of factors and will be discussed later.

## 4. L-LEVEL BANYANS

Next we consider a class of banyans with more regular structure and additional useful properties, but which is still general enough to include most practical designs.

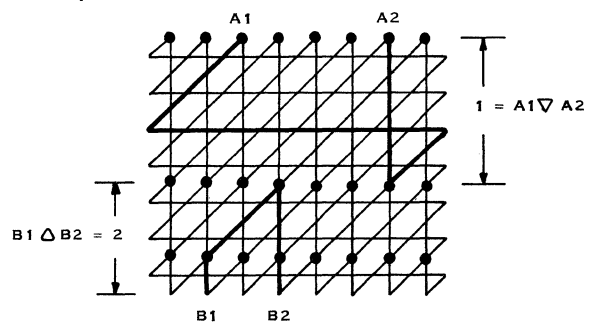
An L-level banyan is simply a banyan whose vertices are arranged in levels so that switches, or arcs of the graph, can only exist between vertices in adjacent levels. For example, the graphs in Figures 3b, 5, and 7 are L-level banyans, but 3a is not. There are actually L+1 levels of vertices in an L-level banyan. They are numbered apexward from 0 to L so that all bases are in level 0 and all apexes are in level L.

Any path from a base to an apex in an L-level banyan has exactly L arcs; thus the propagation time through the network during search and set-up is constant. Moreover, the propagation delay of data through the network cannot exceed that of 2L switches, since in the worst case, data must travel from base to apex to base.

### 4.1 BASE AND APEX DISTANCE

A base distance function,  $B1 \Delta B2$ , can be defined on the bases of any L-level banyan specifying the minimum number of levels up into the banyan a connection must extend to connect two bases, B1 and B2. Similarly, an apex distance function can be defined on the apexes specifying the minimum number of levels down from the top of the structure a connection must extend to connect any pair of apexes. Figure 7 illustrates the concepts of base and apex distance. The darkened paths represent minimal connections. The connection of apexes is presented only as a conceptual aid in ex-

Figure 7. Base and Apex Distance in an L-Level Banyan



<sup>2</sup>This would still require a portion of the control circuitry in a faulty cell to function. A slower search algorithm that avoids this problem has been described by Lipovski (17). Alternatively, a software search algorithm could replace the faster hardware algorithm in the event of hardware failure.

<sup>3</sup>In some cases it may also be possible to connect additional subsystems in a single banyan by rearranging the connections of existing subsystems, but this is only a partial solution and will not be considered further here.

plaining apex distance and would not actually occur in a partitioning network.

The definitions of base and apex distance can be extended to sets of bases and apexes respectively in the same way that point distances are often extended to sets of points. That is, the base distance between any two sets of bases  $B_1$  and  $B_2$  is defined to be the minimum of all distances  $b_1 \Delta b_2$  such that  $b_1 \in B_1$  and  $b_2 \in B_2$ . The analogous extension applies to apex distance.

**Theorem 2:** In an L-level banyan, let  $A_1$  and  $A_2$  be apexes and let  $B_1$  and  $B_2$  be sets of bases. If  $L < (B_1 \Delta B_2) + (A_1 \nabla A_2)$ , then subsystems  $B_1$  and  $B_2$  can be connected without conflict in the same layer with connections rooted at  $A_1$  and  $A_2$  respectively.

**Proof Sketch:** In order for the tree-shaped connection connecting subsystem  $B_1$  with apex  $A_1$  to conflict with that connecting  $B_2$  with  $A_2$ , the two connections must have in common some vertex,  $V$ . But  $V$  must lie in some level  $I$  such that  $B_1 \Delta B_2 \leq I \leq L - (A_1 \nabla A_2)$ . No such  $I$  can exist if  $L < (A_1 \nabla A_2) + (B_1 \Delta B_2)$ .

Theorem 2 not only characterizes a way to avoid conflicts, but also suggests ways to enhance network performance. There are two potentially useful interpretations. First, subsystems close to each other place more stringent requirements on the separation of apexes used than do widely separated subsystems, suggesting that closely spaced subsystems are less likely to be connected in the same layer. Thus, if it is known at design time which resources of a system are most likely to be connected, one might improve performance by gerrymandering the assignment of resources to bases so that bases most likely to be connected tend to be closest. An operating system could also take advantage of this result by allocating closely spaced resource modules to a subsystem whenever possible. The amount of improvement thus obtainable is not estimated here since this would be highly problem dependent, but one can easily contrive extreme examples in which more than one layer would seldom or never be needed.

The second interpretation concerns the selection of apexes. The search procedure described earlier locates all apexes eligible for connecting a new subsystem in a partially occupied layer, but does not determine which of the eligible apexes is the best choice. Theorem 2 now suggests a plausible selection criterion. According to the theorem, any new subsystem can be connected if we can find some apex sufficiently distant from those already in use. Thus apexes most distant from those in use are the most valuable in the sense that they are likely to be eligible for connecting the greatest variety of subsystems. More subsystems might then be connected in a layer by selecting each new eligible apex so as to leave as many "valuable" apexes as possible for subsequent connections. This criterion is ambiguous in some cases, but nevertheless is the conceptual basis for a priority rule found to improve performance in simulated networks. (5)

#### 4.2 FANOUT AND SPREAD

Parameters specifying the number of arcs incident into and out from the vertices of an L-level banyan not only determine the fanout and fanin requirements of circuits used but can also specify its size and shape.

We define a regular banyan to be an L-level banyan in which the number of arcs incident into each vertex

is a constant  $F$  called the fanout and the number incident out from each vertex is a constant  $S$  called the spread. We except, of course, the fact that bases have no arcs incident into them and apexes have non incident out.

Regular banyans would likely be the most economical to fabricate, because they can be built from a number of identical cells, each containing the circuitry associated with a vertex and the arcs incident into it. The fanout and fanin requirements of these cells are determined by  $F$  and  $S$ . The next theorem shows how the number of vertices, and hence cells, in each level of a regular banyan is determined by  $F$ ,  $S$ , and  $L$ , regardless of how the levels are interconnected.

**Theorem 3:** In a regular banyan with  $L$  levels, fanout  $F$ , and spread  $S$ , the number of vertices in any level  $i$  is given by  $N_i = S^i F^{L-i}$ .

**Proof Sketch:** For any given apex, there are  $F^L$  possible paths from various bases. Since there must be exactly one path from each base,  $N_0 = F^L$ . Also, for each  $1 \leq i \leq L$ ,  $N_i = N_{i-1} (S/F)$ . Therefore,  $N_i = F^L (S/F)^i = S^i F^{L-i}$ .

When the fanout of a regular banyan equals its spread, the number of vertices becomes the same in each level. In this case we call it rectangular.

### 5. SPECIFIC BANYAN STRUCTURES

There are two known types of regular banyans of particular interest, SW and CC banyans.<sup>4</sup> Special cases of these structures have been considered previously for a variety of applications.

A structure graphically equivalent to a CC banyan with  $L = 3$  and  $F = S = 4$  has been used in the "Barrel Switch" of the ILLIAC IV Processing Element (19) to shift 64 bits an arbitrary number of places to the left or right.

SW structures were first proposed for partitioning applications by Lipovski (18). Structures graphically equivalent to rectangular banyans with  $F = 2$  had been proposed earlier by Batcher for use as "bitonic sorters". (20) A variety of permutation structures have also been proposed which contain special cases of SW banyans as subgraphs (13-15, 21). Even such common structures as crossbars and homogeneous trees are special cases of SW banyans in which  $L = 1$  and  $S = 1$  respectively.

We are presently concerned with SW and CC banyans as partitioning networks, but this diversity of applications suggest that banyan theory may be useful in other areas as well.

#### 5.1 SW STRUCTURES

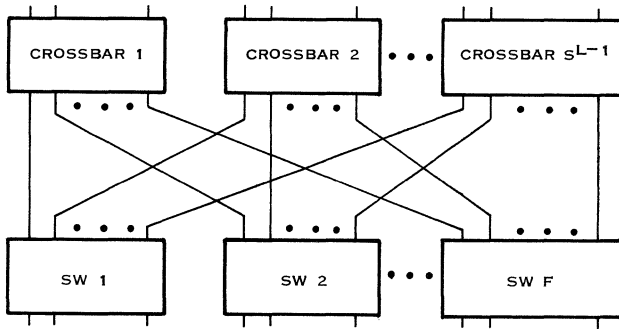
The SW structure is a kind of regular banyan produced by recursively expanding a crossbar structure in the manner of Theorem 1 as illustrated in Figure 8. Examples of SW banyans appear in Figures 4, 5, and 6. The rules for this recursion are as follows:

1) A one-level SW structure with fanout  $F$  and spread  $S$  is simply a crossbar with  $F$  bases and  $S$  apexes.

<sup>4</sup>The term SW has been used in earlier work by Lipovski (18). CC is an acronym for Cylindrical Crosshatch since a CC network can be neatly laid out as a crosshatch pattern on the surface of a cylinder.

2) An L-level SW structure with fanout F and spread S can be synthesized by interconnecting  $S^{L-1}$  crossbars and F identical (L-1)-level SW structures, all with fanout F and spread S. The apexes of the SW structures are connected to the bases of the crossbars such that the interconnection graph is a crossbar. Also we stipulate that each crossbar must be connected to every component SW structure in the same way; i.e., if it is connected to the  $i$ th apex of one SW, it must be connected to the  $i$ th apex of each of the others. The reason for this stipulation will be explained shortly.

Figure 8. Synthesis of an SW Banyan



The base and apex distance functions of an SW banyan tend to group the bases and apexes respectively into nested subsets. It is apparent in Figure 8 that the bases of a synthesized SW banyan may be grouped according to the component SW's above them, forming a partition with F subsets. It is also apparent that any connection between bases in different subsets must be made through one of the crossbars and hence must extend exactly L levels into the network. The distance between two such bases is thus L.

Bases within a subset can always be connected in the component SW above; so when two bases are in the same subset, the distance between them cannot exceed the levels of that component banyan, L-1. To determine whether this distance is equal to L-1 or less than L-1, one can similarly decompose the component SW's and partition each subset into F smaller subsets. Continuing this decomposition, one can obtain L-1 levels of nested subsets such that the distance between any two bases is given by the level of the smallest subset containing both bases.

Similarly, it can be shown that the apex distance function groups apexes into L-1 levels of nested subsets such that each subset is divided into S smaller ones.

As was stated in section 4.1, the base and apex distance functions specify the minimum number of levels into the structure that a connection must extend to connect two bases or apexes respectively. In an SW banyan, these functions also specify the maximum number of levels into the structure that branching may exist in any such tree-shaped connection. The stipulation "each crossbar must be connected to every component SW structure in the same way" is included to insure this property for apex distance.

A consequence of this property is that the converse of Theorem 2 also becomes true making it an if and only if test for conflicts. Thus for the SW structure, the criterion of Theorem 2 not only gives us a way to avoid conflicts but also a characterization of which apexes and bases can and cannot be connected without conflict in a single layer.

## 5.2 CC STRUCTURES

The CC structure is rectangular by definition and thus must have  $S^L$  vertices in each level. Let  $V_i^0, V_i^1, \dots, V_i^{N-1}$  be the vertices in each level

of an L-level CC structure, where  $N = S^L$ . In the graph of this structure, there is an arc from a vertex  $V_k^i$  to a vertex  $V_{k+1}^j$  in the level above whenever  $j = i + mS^k \pmod{N}$  for some  $m = 0, 1, \dots, S-1$ . An example of a CC structure is shown in Figure 7.

To show that this structure is indeed a banyan, we note first that the L-level property insures that the graph contains no loops and hence is that of a partial ordering. To show that there is exactly one path from each base to each apex, consider any such path from an arbitrary base. In propagating from each level k to level k+1, a signal may be shifted  $0, S^k, 2S^k, \dots$ , or  $(S-1)S^k$  places to the right in circular fashion. In propagating through the entire network a signal may then be circularly shifted from 0 to  $S^{L-1}$  places to the right so that there is a possible path to each of the  $S^L$  apexes. Further, since there is an S-way branch at each level, there are exactly  $S^L$  such paths from each base, and hence one to each apex.

The CC structure demonstrates that multi-level regular banyans can be built without using the recursive technique of Theorem 1. Also it can be shown that the base and apex distance functions of a CC banyan differ from those of the recursively defined SW banyans in that bases or apexes appear to be arranged in a circle rather than in nested subsets. The distance between two bases or apexes is then determined by their separation on the circle(5).

## 6. SIMULATION RESULTS

The number of layers typically required for a given banyan to fully partition its bases has not been obtained analytically. To obtain an indication of the layers required, several rectangular banyan networks were simulated.

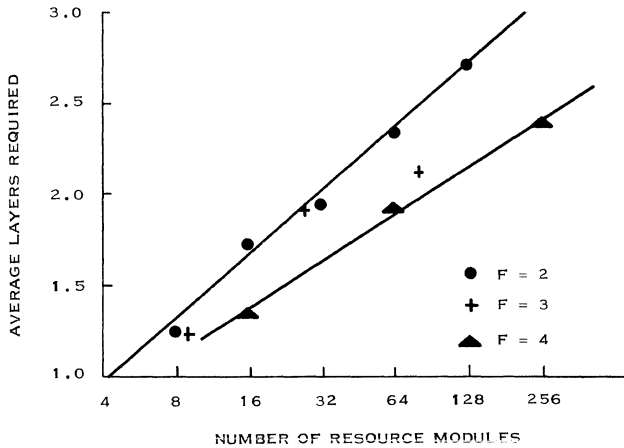
The simulations tested the ability of networks to connect randomly selected partitions. First, the number of subsystems in a partition was selected as a pseudo-random number from 1 to the number of bases in the network. Then each resource module was assigned to one of these subsystems selected at random. The number of modules in any subsystem could thus vary and could even be zero in some cases. Subsystems were then connected one at a time, placing each in the first available layer until the entire partition was realized. All subsystems were then dissolved and the procedure was repeated for a total of 100 partitions. Details of the simulations can be found elsewhere (5).

The average number of layers required to fully connect these partitions was computed for several sizes of rectangular SW banyans, as graphed in Figure 9. With



a fanout of 2 or 4, the average layers required appears to grow logarithmically with the number of resource modules. With a fanout of 3, this function appears to grow more slowly than the logarithm; however, one must be cautious about concluding this with only 3 data points. Larger networks were not simulated because of computer time limitations, but additional simulations of CC networks and of rectangular SW networks with modified setup rules have generally supported the observation that the average number of layers required grows no more rapidly than a logarithmic function of the number of resource modules.

Figure 9. Simulation Results



It is also apparent that with other factors equal, networks with larger fanouts tend to require fewer layers. For example, with 64 bases, the networks in Figure 9 required an average of 2.35 layers with  $F=2$  and 1.91 with  $F=4$ . A similar network with  $F=8$  required only 1.8 layers.

In several respects the results in Figure 9 represent worst case conditions more severe than those likely to be found in actual systems. First, it was assumed that in each partition, every resource module was assigned to some subsystem; i. e., no idle resources. Furthermore, trivial subsystems containing only one module were connected with apexes in the usual fashion even though this would likely be unnecessary in practical systems. These simulations assumed also that knowledge of the base distance function could not be used to enhance performance as suggested in section 4.1.

The priority rule used for selecting apexes in the simulations of Figure 10 is equivalent to selecting the leftmost eligible apex when the network is drawn like that in Figure 6. Additional simulations have shown that some improvement is possible using the criterion suggested in section 4.1.

The average layers required for fully connecting all partitions is a useful performance measure because it indicates how much the maximum allowable data transfer rate available to each subsystem must be degraded when all subsystems use a single multiplexed network. In practical systems, however, it may not be necessary to connect all desired subsystems at once, so that the maximum number of layers used could be limited to a small number. For ex-

ample, in the largest network simulated, an SW with 256 bases and fanout 4, over 87% of the subsystems were connected in the first layer and over 99% in the first two, even though an average of 2.39 and a maximum of 4 layers were required to connect all subsystems. It was similarly found that the other simulated networks could connect most subsystems in a single layer and all or nearly all with two.

## 7. OPTIMUM FANOUT IN RECTANGULAR BANYANS

In this section we will consider two cost/performance functions for rectangular banyans, and will show that for each, there is an optimum fanout which is independent of network size.

It follows from Theorem 3 that there are  $\log_F N$  levels in a rectangular banyan with  $N$  bases. The total number of apex and intermediate vertices is then  $N \log_F N$ . Each of these vertices has  $F$  contacts immediately below, so the cost of the network in contacts is given by

$$C_1(F, N) = F N \log_F N.$$

Since the worst case propagation delay through the network is proportional to the number of levels, the cost delay product is given by:

$$C_2(F, N) = F N \log_F^2 N,$$

This cost/performance measure is especially relevant when resources communicate synchronously, allowing always for worst case delay.

Both functions are of the form

$$C_p(F, N) = F N \log_F^p N.$$

To minimize this with respect to  $F$ , we set the partial with respect to  $F$  equal to zero and solve for  $F$ .

$$\begin{aligned} 0 &= \frac{\partial C_p(F, N)}{\partial F} \\ &= N \log_F^p N + N F \frac{\partial}{\partial F} \log_F^p N \\ &= \frac{N \ln^p N}{\ln^p F} - \frac{N p \ln^p N}{\ln^{p+1} F} \\ \frac{1}{\ln^p F} &= \frac{p}{\ln^{p+1} F} \\ \ln F &= p \\ F &= e^p \end{aligned}$$

Thus the optimum fanouts are  $e$  for function  $C_1$ , and  $e^2$  for  $C_2$ . Optimum integer values are found to be 3 for  $C_1$  and 7 or 8 for  $C_2$ . Note that when  $F = N$  the network becomes a crossbar structure, implying that for large  $N$ , a rectangular crossbar can be considered a nonoptimal special case.<sup>5</sup>

The average layers required for fully connecting random partitions was not considered in this optimization because its dependency on fanout is not precisely known. The simulation results indicate that somewhat fewer layers are required when  $F$  is large, suggesting that optimum fanouts would be somewhat larger if the number of required layers were considered.

<sup>5</sup>The crossbar of Figure 1 is not rectangular since  $S = \frac{N}{2}$  and  $F = N$ , but its cost function still grows as  $N^2$  and exceeds both  $C_1$  and  $C_2$  of optimal banyans for large  $N$ .

One's choice of fanout could also be influenced by such factors as packaging constraints and the fanout capability of devices used. Further, in a regular banyan,  $F$  must be a root of  $N$ .

## 8. CONCLUSIONS

Regular banyan partitioning networks have been described, whose fanout requirements are constant with respect to system size, and whose cost function grows as  $N \log N$  rather than  $N^2$  of the crossbar. Worst case propagation delay grows as  $\log N$ . Disregarding fanout problems, the propagation delay of data paths in a crossbar is constant; however, that of priority hardware used to resolve simultaneous requests within subsystems would still grow as  $\log N$ , assuming methods similar to (17).

Simulation of such networks with up to 256 resource modules has indicated that most subsystems of randomly selected partitions can be connected with only one or two layers, which might prove adequate in many applications.

In applications where the network cannot be thus limited, any partition could be fully connected by a multiplexed network. In the simulated networks, the average layers required to fully connect random partitions appears to grow no more rapidly than  $\log N$ , which still allows a cost/performance advantage over the crossbar for large  $N$ .

The simulation results presented here indicate that the number of layers required can be small enough not to offset the cost advantages of banyans in large systems. The networks were simulated under artificial conditions that were worst case in several respects. Many variations of banyan networks are possible, only some of which have been presented here. It would indeed be interesting to apply a banyan network to a specific system where it could be tailored to requirements.

This paper has concerned itself with the use of banyan networks for partitioning applications. Consequently, we have not attempted to compare cost performance with networks designed for different functions, such as permuting or store-and-forward message switching. It is felt, however, that the adaptation of banyan structures for such applications warrants further study.

Theoretical results concerning the behavior and structure of banyans can provide insight and suggest ways to enhance performance. With increased notational complexity, most of the theoretical results discussed here for regular banyans, including SW and CC structures, can be extended to L-level banyans in which fanout and spread may be different for each level (5). Since a number of structures proposed previously for other applications are special cases of banyans or contain them as subgraphs, it is expected that banyan theory could also be useful in other areas, especially that of permutation networks.

## REFERENCES

1. H.B. Baskin, E.B. Horowitz, R.D. Tennison and L.E. Rittenhouse, "A Modular Computer Sharing System," Communications of the ACM, Vol. 12, No. 10, pp. 551-559, Oct., 1969.
2. H.B. Baskin, B.R. Borgerson, and R. Roberts, "Prime - A Modular Architecture for Terminal-Oriented Systems," AFIPS Proc. SJCC, Vol. 40, pp. 431-437, 1972.
3. W.A. Wulf, and C.G. Bell, "C.mmp - A Multi-mini-processor," AFIPS Proc. FJCC, Vol. 41, pp. 765-777, 1972.
4. G.W. Schultz, R.M. Holt, and H.L. McFarland, "A Guide to Using LSI Microprocessors," Computer, pp. 13-19, June, 1973.
5. L.R. Goke, Connecting Networks for Partitioning Polymorphic Systems, Doctoral Dissertation, University of Florida, under preparation.
6. PDP-11 Handbook, Digital Equipment Corporation, Maynard, Mass., 1969.
7. J. Basiji and A.B. Bergh, "Central Bus Links Modular HP 3000 Hardware," Hewlett-Packard Journal, pp. 9-14, Jan., 1973.
8. W.E. Vice, A.J. Brodersen, G.J. Lipovski, "On Integrated Circuit Bidirectional Amplifiers," accepted for publication in Journal of Solid State Circuits, Oct., 1973.
9. V.E. Benes, "Algebraic and Topological Properties of Connecting Networks," Bell System Technical Journal, pp. 1249-1273, July, 1962.
10. J.T. Quatse, P. Gaulene and D. Dodge, "The External Access Network of a Modular Computer System," AFIPS Proc. SJCC, Vol. 40, pp. 783-790, 1972.
11. J.P. Anderson, Samuel A. Hoffman, J. Shifman, and R.J. Williams, "D825 - A Multiple-Computer System for Command and Control," AFIPS Proc. FJCC, Vol. 22, pp. 86, 96, 1962.
12. R.E. Porter, "The RW 400 - A New Polymorphic Data System," Datamation, Vol. 6, No. 1, pp. 8-14, Jan./Feb., 1960.
13. L.J. Goldstein and S.W. Leibholz, "On the Synthesis of Signal Switching Networks with Transient Blocking," IEEE Transactions on Electronic Computers, Vol. EC-16, No. 5, pp. 637-641, Oct., 1967.
14. A. Waksman, "A Permutation Network," Journal of the ACM, Vol. 15, No. 1, pp. 159-163, Jan., 1968.
15. A.E. Joel, Jr., "On Permutation Switching Networks," Bell System Technical Journal, pp. 813, May/June, 1968.
16. Claude Berge, The Theory of Graphs, p. 12, John Wiley and Sons, Inc. New York, 1962.
17. C.C. Foster, "Determination of Priority in Associative Memories," IEEE Transactions on Computers, Vol. C-17, No. 8, pp. 788-789, Aug., 1968.
18. G.J. Lipovski, "The Architecture of a Large Associative Processor," AFIPS Proc. SJCC, Vol. 36, pp. 385-396, 1970.
19. R.L. Davis, "The ILLIAC IV Processing Element," IEEE Transactions on Computers, Vol. C-18, No. 9, pp. 800-816, Sept., 1969.
20. K.E. Batcher, "Sorting Networks and their Applications," AFIPS Proc. SJCC, Vol. 32, pp. 307-314, 1968.
21. V.E. Benes, "Optimal Rearrangeable Multistage Connecting Networks," Bell System Technical Journal, pp. 1641-1656, July, 1964.

# STRUCTURE OF DIGITAL SYSTEM DESCRIPTION LANGUAGES

Harry F. Jordan

Burton J. Smith

*Electrical Engineering Department  
University of Colorado*

## Abstract

Several languages have been developed for or applied to the problem of describing digital hardware systems. This paper points out some of the problems encountered in hardware descriptions, particularly where they are distinct from concepts appearing in programming languages.

## Introduction

There are three major goals of a hardware description language: human comprehension, simulation and construction. The requirements imposed by these goals are most easily specified in reverse order. The goal of system construction requires that the language specifically describe the actual hardware needed to build the machine. The language need not, however, describe the structure of any sub-unit which is available as one piece, such as an MSI or LSI integrated circuit. A description of the terminal behavior of such sub-units may be necessary but their actual construction is not of interest to the designer. It must be clear from the description where hardware is implicitly specified in a description, as in the case of multiplexers on register inputs when the register may receive information from several sources.

The goal of simulation requires an accurate behavioral input-output description of each sub-unit in the machine. The behavioral description of sub-units need not be in any correspondence with the structure of the sub-units, but it must be in a form which is executable by the simulator. A simulation may be required to produce more or less detailed results and therefore the structural description of the circuit might be carried out to different depths before the behavioral type of description is employed. In all cases, however, one must be able to reduce every element of the description of a system to a behavioral description in terms of the language of the simulator.

The goal of human comprehension is somewhat more difficult to define. This goal is first in order of importance because the utility of any language depends on how easily human designers comprehend and write descriptions in the language. As Iverson has pointed out in describing APL [1], effective suppression of inessential detail is important to human comprehension and ease of use. However, the design environment in which hardware descriptions are done is quite variable. For example, detail which is nonessential to system structure if a large scale integration arithmetic and logical unit is to be employed becomes essential if

the unit is to be constructed out of smaller sub-units. In such variable environments, the effective suppression of detail seems to depend upon flexible mechanisms for implicit substructuring such as are afforded by extensible languages. In such an extensible language, concise syntactical and semantic constructs can be defined and later used in a simple form in the description of the system or a set of systems based on the same hardware primitives.

The authors feel that the syntactic structure of a language is important to human comprehension in so far as it reflects the logical structure of the thing described. The sequencing of statements, block structuring, if-then-else clauses, and iteration clauses are syntactic features of programming languages which directly reflect execution time features of the computation described by a program written in the language. One of the important problems in designing high level languages for describing digital systems is that of isolating significant logical structure features of systems and providing syntactic constructs in the description language which accurately reflect these features.

## Parallelism

Several semantic problems arise in describing the structure and operation of a digital computer or other digital system. The primary problem is that of describing parallelism in the operation. Digital systems consist of a large number of components connected in a complex way and operating sequentially in time. In order to successfully describe such a system one must be able to group elements that are logically connected to one another in the electronic circuitry and to describe the operation of this sub-unit consisting of a set of connected circuits. One must also be able to associate groups of steps which take place sequentially in time to describe a time-sequence or sub-sequence of operations within the computer. The necessity for grouping elements both in space and in time gives rise to the primary linguistic problems of describing a digital computer. We thus wish to consider what properties a descriptive mechanism or language must have in order to successfully describe digital computers.

Iverson, Falkoff and Sussenguth have used the APL language to describe the hardware of the IBM 360 series of computers. [2] The primary advantage of APL in describing a digital computer stems from the fact that it has a large number of primitives which specify

inherently parallel operations. The primitives involved are primarily operations on bit vectors. The APL primitives have the ability to transfer the values of bit vectors from one variable or register to another, obtain values from subfields of large bit vectors, and apply certain transformations to the bit vectors either one bit at a time or over all bits of the vector. These concepts are quite natural to a parallel computer. They are somewhat less applicable to serial computers. The structure of the APL primitives is parallel in nature, but the overall structure of the language is sequential. Programs in APL consist of a sequence of consecutively numbered and executed steps as in most other programming languages. It is to be expected then that the points at which APL becomes strained in describing computer hardware are just those points at which large scale parallelism becomes a factor in the design. Sub-units which have internal sequential structure yet operate in parallel in a machine are not handled smoothly by APL.

A higher degree of flexibility in describing parallel and sequential operations is afforded by the ISP language developed by Bell and Newell.[3] By using the semi-colon and the semi-colon followed by "next" properly in an ISP description and by including parentheses in appropriate places a complex structure built of sequential and parallel sub-units can be constructed. The technique is quite similar to describing a series-parallel electrical network. The types of structures which cannot be described with the ISP type mechanism of a parallel separator and a sequential separator are in fact just those cross-linked type structures which correspond to bridge-type connections in an electrical circuit. The most general case of course is a group of nodes representing elementary actions with a partial ordering imposed on the nodes. It seems that none of the familiar syntactic mechanisms form structures similar to general partial orders. (On the other hand much execution sequence information is clearly mirrored in programming language syntax.) The cross-linked structures do not often appear in computer design, and a language which offers only the series-parallel mechanism of description will be quite adequate for a large number of applications.

The syntactic structure of this mechanism can be summarized in BNF as follows:

```

<system description> ::= <step>|
  <system description><sequential separator><step>
<step> ::= <action>|
  <step><parallel separator><action>

```

<action> ::= <elementary action>|(<system description>)

An example using : as the parallel separator, → as the sequential separator and EA<sub>i</sub> as a name for the *i*th elementary action is:

EA1 → EA2 → EA3: EA4: EA5 → EA6: (EA7 → EA8) → EA9

The partial ordering imposed by the above description can be represented by the covering relation diagram in figure 1.

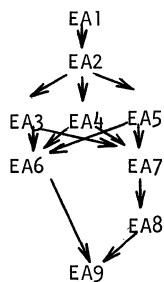


Figure 1

The situation shown in the partial order diagrammed in figure 2 where EA2 and EA3 must be complete before EA4 starts whereas starting EA5 requires only the completion of EA2 cannot be represented by the above linguistic mechanism.

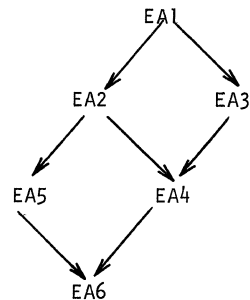


Figure 2

The closest approximation imposes the restriction that EA3 precede EA5 which is not required by the original structure.

Notice that there is no timing information present in the above descriptions. Only sequence information is given as a set of requirements on which actions precede others. Timing is more explicit in a language such as Schorr's Register Transfer Language.[4] Using the conditional execution feature of Schorr's language the sequence information inherent in EA1 → ((EA2 → EA4): EA3) → EA5 might be expressed by:

$|t_1| : EA1 ; 1 \rightarrow t_2 ; 1 \rightarrow t_3$

$|t_2| : EA2 ; 1 \rightarrow t_4$

$|t_3| : EA3 ; 1 \rightarrow t_{5B}$

$|t_4| : EA4 ; 1 \rightarrow t_{5A}$

$|t_{5A} \wedge t_{5B}| : EA5$

The timing is more explicit in the RTL but extra detail (the arbitrary ordering of the lines and names for the times) tends to obscure the sequence information. It seems that the specification of sequence of actions in a digital system may be a higher level concept than the specification of timing. Note that when a high level system description specifies only a partial order on actions the tasks of simulation and construction are complicated by the need for implicit rules for resolving ambiguities in timing.

### Function Types

Another type of descriptive dichotomy which exists in computer hardware description is the need to describe sub-units in several different ways. Linguistically, a sub-unit may be described in a single statement or by a procedure definition, and consists of a set of input variables and a set of output variables together with some well-defined set of rules for computing the values of the output variables from the values of the input variables. Two distinct types of functions arise in the description of hardware: combinational functions and sequential functions. The combinational functions may be thought of as statically defined structures in the sense that as long as the inputs are constant the output is constant (except for propagation delay effects), and inputs and outputs are quite distinct. The sequential function, on the other hand, has a set of parameters associated with it which can be thought of as registers. The sequential function is invoked at a particular point in time; it uses the values in the input registers to perform some computation in some finite number of steps and produces results in the

output registers, some of which may be the same as the input registers.

There are also two distinct types of function usage. One sort of use involves assembling a separate set of hardware according to the specifications set forth in the function definition. In this case, the definition may be thought of as generic in nature, describing many devices of the same structure. The second function usage involves using a single hardware unit at several places within a system description with the separate uses of the unit occurring at different times and perhaps involving multiplexed inputs and/or outputs. An example of a generic use of a function definition would be to describe the structure of several similar 16 bit counters in terms of their components. Multiple usage of a specific function definition would occur if the same adder were used for arithmetic and effective address computation.

One possible way to dissolve this descriptive dichotomy is to relegate generic function descriptions to a strictly linguistic mechanism such as text substitution macros. If this is done then the appearance of a function name with appropriate parameters for input and output can be thought of as simply a name for the computation which takes place within the function definition. Specific functions, on the other hand, can be represented as procedures or closed subroutines which are invoked during the running of the system. Each computation begins at the point in time at which the procedure is invoked, and results are available after the characteristic delay time associated with the sequential device or the system of combinational logic and multiplexers.

It is convenient to allow a sequential function to have multiple entry points to clarify the correspondence between structural and sequential type descriptions. This facility permits the description of sub-units which perform several functions. An example of such a sub-unit is a shift register which may be shifted left by clocking one input, shifted right by clocking another input, or loaded in parallel by clocking a third input. Each of these three clock inputs can be represented by a distinct entry point.

### Control

Yet another dichotomy exists in hardware description, namely the dichotomy between data and control. It is important for the suppression of detail in a high-level description to let the control signals be implicitly described by the order in which statements are to be executed, but there must be mechanisms for interaction between control and data. After an instruction has been decoded, for example, the signals which represent the instruction must cause a transfer of control to the portion of the description which executes that instruction. This is conventionally handled by conditional statements of one kind or another. It is also useful to allow control signals to be treated as data; this can be done by introducing the object "1\*" which is logical 1 when the statement containing the "1\*" is being executed and logical 0 otherwise. This concept is similar to that of a program counter in a programming language, but due to parallelism there may be more than one of them active at a given time. This facility can be used within a language to deal with the problem of implicit multiplexing, as follows. The language associates with each register R in the description two expressions: INPUT (R) and CLOCK (R). These expressions are obtained by initially setting each of them to 0 and then examining every statement in the description; whenever a register transfer

$$R \leftarrow S$$

is encountered, INPUT (R) is replaced by

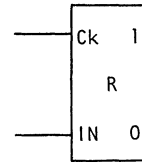
$$\text{INPUT (R)} \uplus S *$$

and CLOCK (R) is replaced by

$$\text{CLOCK (R)} \uplus *$$

Here  $\uplus$  is an operation equal to  $a \vee b$  if  $a \wedge b = 0$  and is undefined otherwise. A simple example of this mechanism is shown in figure 3.

### Translation of Register Transfer Statements to Networks



Translation of  $R \leftarrow S$

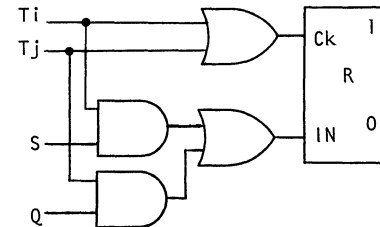
$$\text{INPUT (R)} = \dots \uplus S \wedge 1^* \uplus \dots$$

$$\text{CLOCK (R)} = \dots \uplus 1^* \uplus \dots$$

where  $a \uplus b = a \vee b$  if  $a \wedge b = 0$  and is undefined if  $a \wedge b = 1$

#### TIME ACTION

$T_i$	$R \leftarrow S$
$\vdots$	$\vdots$
$T_j$	$R \leftarrow Q$



$$\text{INPUT (R)} = T_i \wedge S \vee T_j \wedge Q$$

$$\text{CLOCK (R)} = T_i \vee T_j$$

$$\text{Validity Condition } T_i \wedge T_j = 0$$

Figure 3

### Levels of Description

A hardware description language should be applicable to various levels of description. In particular, it is extremely useful at any level to have the ability to describe the input/output behavior of some "black box" circuit without describing its internal construction. In this way, portions of the circuit may have their descriptions put off until a lower level of description is reached. Such a description of the input-output behavior of a "black box" should be put in the clearest and most convenient form possible. In some cases, this may mean that the input-output description is a sequential description when in fact the unit is a combinational unit, or it may mean that the description may be combinational while the unit actually operates sequentially. In general, then, it is not desirable to require that the structure of the box match that of the description, since this internal structure is precisely what we are trying to suppress. There is probably no need for a separate language for the description of the terminal behavior of sub-units. The hardware description language itself should be flexible enough to provide a clear and concise description of any possible unit. There should, however, be some distinction made between the two kinds of application of the language to clarify whether it is being used to describe the actual structure of a sub-unit of a machine or merely to specify the input-output behavior of the sub-unit for the purpose of describing the activity of the rest of the machine.

There must also be provided methods for describing timing and sequence of sub-unit interfaces when this information is not reflected by the behavioral description.

## Use of Names

We also wish to consider the role of names in a hardware description language. There are three classes of objects which may be named in the description of a machine. These three classes have somewhat different properties. One use of names is in the description of values stored in registers of the machine. These names play roles quite similar to the roles played by variable names in programming languages. The values associated with the names can be non-destructively read and used, and are changed only as a result of an action which stores a new value into the register. Another possible use of names in describing computer hardware is to identify Boolean signals or vectors of Boolean signals which appear within the machine. For example, consider a bus within a machine which is used to transfer values among the registers of the machine. The value on the bus may change either because of a change in the contents of the register that is currently multiplexed onto the bus or because of a change in the contents of one or more of the flipflops that determine which register is multiplexed onto the bus. The value of the bus is therefore a combinational function of the values of several registers. It is useful to name the bus in order to specify transfers of values between the bus and registers; such a usage of a name illustrates the second role for names. Finally, names may be used to designate system modules (either generic or specific) the behavior of which involves a mixture of both registers and signals.

The usages for names discussed above can be distinguished by declaration. If all registers in the machine must be declared and all combinational functions are declared then the digital system is well defined for the purposes, say, of simulation. The simulator can maintain internal variables which keep track of the current values of each register, and whenever the simulator encounters the use of a combinational function output as a value it can examine the necessary combinational function definitions to determine this output as a function of the values of the internal variables. Of course it may be necessary for the simulator to trace back through several levels of combinational function definitions in order to find registers whose values completely determine the final output. A summary of three possible variants of the assignment statement in a programming language based on the different declarations and use of names mentioned above is given in figure 4.

## BIBLIOGRAPHY

1. Iverson, K.E., "A Programming Language," John Wiley and Sons, N.Y., 1962.
2. Falkoff, A.D. and Iverson, K.E., "A Formal Description of System/360," *IBM Systems Journal*, Vol. 3, No. 3, 1964.
3. Bell, C.G. and Newell, A., "Computer Structures: Readings and Examples," McGraw-Hill, N.Y., 1971.
4. Schorr, H., "Computer Aided Digital System Design Using a Register Transfer Language," *IEEE Trans. on Electronic Computers*, Vol. EC-13, pp. 730-737, Dec., 1964.

## DECLARATION AND USE OF NAMES

1. Signal X  
 $X @ \text{FCN}(Y,Z)$       X is wired to output of FCN
2. Signal X  
 $X := \text{FCN}(Y,Z)$       The value of X is made to match the output of FCN for the duration of this step.
3. Register X  
 $X \leftarrow \text{FCN}(Y,Z)$       The value of FCN is strobed into register X in this step.

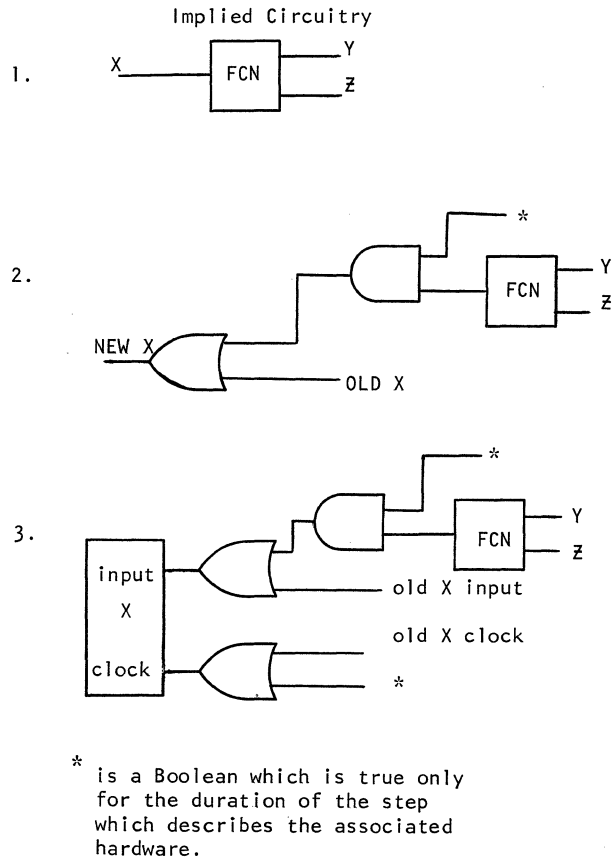


Figure 4

# VDL—A DEFINITION SYSTEM FOR ALL LEVELS

John A. N. Lee

*Professor of Computer Science  
University of Massachusetts at Amherst*

## ABSTRACT

The VDL system for the description of programming languages which was originally used for the definition of PL/I is extended to the description of processors. This paper shows the relationship between the language of definition and the abstract machine over which the semantics of the language are specified. It is demonstrated that the level of description can be chosen to suit the various needs of the computing community, each level being well nested within its outer level, whilst using only one language of definition.

From the point of view of processor design, indications are given of the means by which a description can be transformed into an implementable system of data paths, registers and drivers.

## INTRODUCTION

The techniques of formal definition as applied to the description of the PL/I programming language by Lucas and Walk (Lul), has since been applied to other systems by the author (Le1,Le2). On the basis that the definition of a programming language consists of a system of definitions of algorithms, the method of definition is applicable to not only languages, but also the description and definition of algorithms, in particular, to processors.

The formal definitional system described by Lucas and Walk consists of a synthetic language defined to operate over a set of data objects which can be described in terms of non-cyclic trees. In the definition of a programming language such as PL/I, it is necessary to consider not only the definition of the syntax of the source language and a description of the technique for converting that language (or an analyzed version of it) into a form suitable for use in the description of its semantics.

In the case of describing a processor, we shall pay little attention to the syntactic form of the associated machine language and no attention at all to the external form of that language. In fact, we shall assume that any program to be executed exists only within the object which represents, in the abstract machine which models the prototype, the storage part of the prototype.

The VDL definitional schema is so organized, as will be described later, that the techniques of top-down programming naturally evolve and thus the level of description can be matched with the needs for understanding of the intended recipient of the description. Further, by a judicious choice of identifiers in the description, the understanding of the recipients can be enhanced to the point where the description is highly readable. Using the macro-expansion techniques of description which are analogous with the commonly used techniques of describing machine instructions in terms of lower level actions associated with event times, a single description can contain a continuum of definition levels. At the outer level, the description can correspond very closely to the style of description which is associated with a machine reference manual. At succeeding levels of definition, more detailed descriptions can be offered which reveal further details of implementation. For example, the outer level of definition may reveal that (say) an ADD instruction is executed by adding the contents of the

accumulator and contents of the referenced cell, and then leaving the result in the accumulator. For most purposes of programming this definition will be sufficient; however, the further description of these components can show the utilization of the individual registers and the data paths between the registers. This level may not necessarily reveal the actions of the drivers for the registers, this being left to the next level, until eventually the logic level of the gates is reached. This ability of a single description language to provide these many levels of definition makes it a prime candidate for the general usage as processor descriptor. That is, rather than having several languages for the description of each level of a processor action, the Vienna Definition Language, by its design, provides for all the definitional needs of the computer architect.

The major emphasis of the usage of VDL has been on the linguistic aspects of the definitional schema, little attention having been drawn to the abstract machine, the actions of which the language describes. The properties of this abstract machine are only now being investigated more thoroughly and it can be expected that these investigations will provide a firm basis for the development of the properties of the machine described.

Within the definitional scheme itself there exists two levels of abstraction: the level of description used previously in the semantics of programming languages, and an inner machine, being a finite state machine over which the "outer level" descriptors are defined.

## THE INNER MACHINE

A Definition Machine is a 5-tuple  $\{\Sigma, \emptyset, P, \mu, \tau\}$   
where  $\Sigma = S \cup \{I\}$

- S is a finite non-empty set of closed one-to-one mapping functions (called selectors or selector functions) over  $\emptyset$ ,
- I is the identity selector or function,
- $\emptyset$  is a finite non-empty set of objects,  
where  $\emptyset = CO \cup EO$ , and
- CO is a finite non-empty set of composite objects,
- EO is a finite non-empty set of elementary objects;
- P is a finite (possibly empty) set of predicates,
- $\mu$  is the mutation operator, and
- $\tau$  is the search function.

Objects in  $\emptyset$  are defined formally to be a finite non-empty set of unique pairs  $\langle s:A \rangle$  which specify the range ( $A \in \emptyset$ ) of each selector function (s) in the set S over the domain of the object (B). Notationally, an object identified as B is represented as

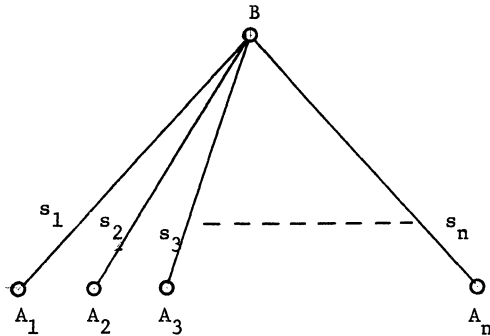
$$B = \{\langle s_1:A_1 \rangle, \langle s_2:A_2 \rangle, \dots, \langle s_n:A_n \rangle\}$$

where  $\{s_1, s_2, \dots, s_n\} = S$  and  $(\forall i) (A_i \in \emptyset)$  and  $B \in \emptyset$ .

For simplicity, all pairs whose second component is the null object are normally omitted from this set. The set of unique pairs which specify the range of each selector in S over the object is called the characteristic set of the object. The application of a selector function to an object is symbolized by  $s(B)$ . If  $B = \{\dots, \langle s_i:A_i \rangle, \dots\}$ , then by definition above,  $s_i(B)$  yields  $A_i$ .

For the purposes of description, these characteristic sets of objects have been likened to non-cyclic trees, and thus the common representation of an object is as a tree shown in Figure 1.

FIGURE 1  
A TYPICAL COMPOSITE OBJECT



Since the objects selected by the selector functions from an object are themselves (by definition) objects, then the repeated application of selector functions is equivalent to a walk through the tree representation from the root to the root of some subtree. This repeated application of selectors leads to the usage of composite selectors.

A composite selector  $K$  is the representation of the successive application of selector functions to an object.

If  $K = s_1 \dots s_n$ , then

$$s_1 \cdot s_2 \cdot \dots \cdot s_n(X) = s_1(s_2(\dots(s_n(X))\dots))$$

Df

where  $(\forall i) (s_i \in S)$  and  $K \in S^+$

As a matter of nomenclature, the selector function  $s_i$  ( $\in S$ ) is known as a simple selector.

The object selected from a composite object by the composite selector  $K$  is known as the  $K$ -component.

An elementary object within the machine ( $eo \in EO$ ) is characterized by a set in which the range of every selector is the null object ( $\Omega$ ). Elementary objects may be regarded as "atomic" or "indivisible" objects. The precise set of elementary objects associated with a definition must be defined in advance and may be dependent on the level of definition. For example, in the case of a user level of definition it may be sufficient to consider the set of elementary object to be words, whereas for the gate level of definition the set of objects may simply be the binary digits. This definition of an elementary object then provides a simple definition of a composite object:

A composite object is an object in which the range of at least one selector function  $s \in S$  is not the null object.

$$(\exists s)(s(B) \neq \Omega), s \in S, B \in CO \subset \emptyset$$

The primary function which operates over objects is the mutation function  $\mu$  which is a closed function over the set of objects  $\emptyset$ . Notationally, the function and its arguments are represented by

$$\mu(A; \langle s; B \rangle)$$

The range of the function is

$$(A - \{ \langle s; s(A) \rangle \}) \cup \{ \langle s; B \rangle \}$$

That is, the mutation function creates a copy of object  $A$  (the subject argument) in which the  $s$ -component is replaced by the object  $B$ . This elemental function has the property that three basic operations can be simulated by its usage: replacement, deletion and construction. As described above, the basic operation of replacement is obvious; object  $B$  replaces the previous  $s$ -component in the copy of the object  $A$ . By specifying that the replacement object is the null object ( $\Omega$ ), then the process of deletion of simulated. Similarly, if the original subject argument ( $A$ ) had been the null object, then any mutation of that object with non-null objects constructs a new object.

The set of predicates in the inner machine provides a basis for the discriminating properties of the definitional schema. In the definition of programming languages, predicates are used to define the valid objects which can compose the abstract text (c.f., abstract syntax (Mcl)) over which the semantics of the language are to be defined. In a processor, these predicates describe the internal structure of the machine being modeled and certain properties which it is necessary to have the capability of recognizing, such as that the contents of the accumulator are zero. Combined with expressions, predicates form conditional expressions of the form

$$p_1 \rightarrow e_1, p_2 \rightarrow e_2$$

which can be defined by the logical expression

$$p_i \ \& \ (\forall j < i) (\neg p_j) \supset e_i$$

These expressions, in the general case, result in an undefined value if none of the predicates are true. Whereas this is advantageous when the subject of the description is a programming language and there can exist some "undefined" situations, but in the case of a processor, these conditions should be closed properly. Considering the levels of definition discussed before, conditional expressions correspond closely to the gate level of description. The search function ( $\tau$ ) did not originally exist in the Lucas and Walk (LU1) descriptions and definitions, but has been added by the author (LE1) to provide more generality. The Lucas and Walk unique selector function ( $\iota$ ) is simply a special case of the search function. Further, the search function closely resembles the associative memory polling operation and provides a sound basis for the simulation of set operations in language descriptions.

The search function  $\tau$  selects from  $\emptyset$ , a set of objects, each member of which conforms to the specified predicate  $is\text{-}pred$ .

$$(\tau x) (is\text{-}pred(x)) = \{x | x \in \emptyset \ \& \ is\text{-}pred(x) = T\}^\dagger$$

The expression  $(\tau x)(is\text{-}pred(x))$  is read as "the set of those objects ( $x$ ) chosen from  $\emptyset$  such that the predicate  $is\text{-}pred$  is satisfied."

#### THE OUTER MACHINE

Using the properties defined in the preceding section, we may now devise a definitional model, which will be the basis for describing processors. This finite state machine contains a set of states which contain information on the data being manipulated and the instructions (or programs) which define the transformations to be executed over the data, and a function (the State Transition Function) will interpret and execute the instructions in the current state of the machine.

<sup>†</sup> Using a standard set notation.



In attempting to define the properties of a processor, the state of the machine is defined to contain, as one of its components the complete set of registers and storage devices of the processor being modeled. Since the definition is itself a program, then the instructions which reside in the storage part of the processor being modeled act as data elements. In the succeeding description here, we shall reserve the term instructions to refer to the instructions contained in the definition machine.

Within the state of the definition machine there exists a special component which contains the set of instructions which are awaiting execution, and which by their execution will represent the execution of the commands in the processor being modeled. This component is known as the control stack and can easily be represented by a regular VDL object. However, for the purposes of description we can regard the control stack to be a tree in which the definitional instructions are contained as the nodes of the tree (c.f., the VDL object represented as a tree, in which objects exist only as the leaves of the branches).

By Lucas and Walk (LU1) the order of execution of the definitional instructions is defined to be restricted to any one of the instructions which exists at a leaf of the control stack. Since the execution of instructions (see later) includes their removal from the control stack, this provides a multi-stacking facility whereby instructions can be inhibited from execution until all other instructions on their branch (in their stack) have been executed. Whilst Lucas and Walk insisted that any one of the candidate instructions can be executed during a state transition cycle, this concept is extended here so as to provide for the asynchronous execution of all instructions which are existing at the leaves of the control tree. This process adequately simulates the asynchronous operations within a processor, but solves none of the problems of race conditions which are thereby possible. However, since the definition of instructions requires explicit reference to any data assignments and there exist no side effects within VDL, there exists a clear potentiality for proving that race conditions either exist or are non-existent.

The initial state of the definition machine is one of the elements of the definition of each processor. This may correspond directly to the conditions which are existing at the time that the manual actions of depositing an address into the program counter and depressing the RUN key are performed. A final state (a halting state) of the definition machine is the state in which the contents of the control stack is null; that is, there are no further definitional instructions to be executed. Other final states may include cases where some error condition has arisen and the execution of the instructions existing in the stack is undefined.

Definitional instructions can be executed (depending on conditions existing within the state of the machine) either as macro-expansion instructions or as state-modifying instructions. In the former case, the execution of the instruction has the effect of replacing itself by a new instruction subtree thereby simulating either the passage from one level of definition to the next or the sequencing of operations. In the case of state-modifying execution, the effect is to mutate the state of the machine (other than the control stack) thereby simulating operations over the registers in the prototype, and then to remove that instruction from the control stack.

Whilst there is only one style of execution that an instruction be subject to at the time of its execution,

the definition of instructions can specify varying styles depending on the conditions existing at the instant of execution of the instruction. Thus a definitional instruction may have several definitions itself, only one being applicable at any time. These individual definitions are termed "groups."

The means by which definition groups are chosen from within the general instruction definition set is a conditional expression, the right hand sides of which are the definition groups. That is, the general form of an instruction definition is

$$\begin{aligned} \text{inst}(q_1, \dots, q_n) = \\ p_1 \rightarrow \text{group}_1 \\ \dots \\ \dots \\ p_m \rightarrow \text{group}_m \end{aligned}$$

where  $q_1, \dots, q_n$  are parameters which are replaced by the values  $q_1$  of  $q_n$  the arguments specified in the instruction at the time that the instruction is placed into the control stack,  $p_1, \dots, p_m$  are predicate expressions which are functions of the set of parameters  $q$ , system defined predicates and the state of the machine. It will be shown later that in the case of describing processors at the register level, the set of parameters (and consequently the corresponding set of arguments) is unnecessary, the need for a parameter showing the need for a register in the prototype.

Where the group is to be a macro-expansion definition, the notation is to show not only the set of instructions which are to replace the instruction being executed in the control stack, but also the structural relations between those instructions. The notation contains two basic rules for demonstrating the nodal position of instructions within the tree:

- i) indentation indicates a lower level of tree placement (lower in the sense of movement between the root at the top and leaves at the bottom) than instructions not as deeply indented.
- ii) punctuation indicates either a continuation of a level by the use of a comma (,) or completion of a level by the use of a semicolon (;) except where the instruction is the last in the group when no punctuation is needed.

It is important to note that since the order of execution of instructions is from the leaves of the tree toward the root, then the instruction(s) at the bottom of a group representation are the earlier candidates for execution. Normal sequential execution of a group of instructions is represented by a diagonal sequence of instructions separated by semicolons:

```

inst-1;
inst-2;
inst-3;
inst-4

```

This set of instructions would be executed in the order

```

inst-4 inst-3 inst-2 inst-1

```

A single instruction cannot be replaced by a set of asynchronous instructions since such a set does not form a proper tree structure. Instead a simple one level tree with one root must be formed. In essence this corresponds to the case where a number of instructions can be executed simultaneously and the execution of a succeeding instruction must await their completion. The root instruction in this group then acts as a semaphore since it prevents the execution of

instructions higher on the same branch until it is cleared. Such a group of instructions is represented in the form

```

inst-1;
  inst-2,
  inst-3,
  inst-4

```

In this group, the instructions inst-2, inst-3 and inst-4 can be executed asynchronously (for our purposes here) but inst-1 cannot be executed until all of those instructions have run to completion.

State-modifying definition groups specify the changes to be made to the state of the machine (with the exception of the control stack). Each group corresponds closely to a mutation operation, the subject argument of the mutation being the state of the machine. Thus the definition group is a listing of the selector:value pairs, the selectors being applicable to the state of the machine and the values being functions over the parameters (replaced by the argument values) of the instruction (if any) and components of the state of the machine. The general form of a state-modifying group is

```

s-sc1:exp1
...
...
s-scm:expm

```

where the  $s-sc_i$  are selector functions and  $exp_i$  are evaluated to  $v_i$  the values which are to be placed in the state. By the judicious choice of selector names, the data paths in the processor can easily be simulated. For example, let us assume that the memory address register is represented as the  $s-mar$  component of the state ( $\xi$ ) and that the program counter is represented as the  $s-pc$  component. Then the operation of transferring the contents of the program counter to the memory address register can be represented by the definitional instruction pc-to-mar and be defined simply by

```

pc-to-mar =
  s-mar:s-pc( $\xi$ )

```

which states:

"Replace the contents of the  $s-mar$  component of the state by the contents of the  $s-pc$  component of the state."

Since we are dealing with a finite state abstract machine, the question of timing between the acquisition of the data elements of an operation and the placement of the result in the state is overcome by the simple ruse that the new state is a copy of the old state. Thus the execution of an elementary shift command (over a three bit register) is well defined:

```

shift =
  bit-0*s-acc:bit-1*s-acc( $\xi$ )
  bit-1*s-acc:bit-2*s-acc( $\xi$ )
  bit-2*s-acc:bit-0*s-acc( $\xi$ )

```

In this definition, the selector functions are composite, the accumulator being represented by the  $s-acc$  component of the state and the individual bits within the accumulator being selected by the functions of the form  $bit-i$ . That is, the functional composition operator ( $\cdot$ ) can be read as "of". Since it will be necessary to reference elements of state components in a generalized form, we shall permit the extension of the explicit naming of selector functions to include a

functional notation in which the index of selection is included as an argument. For example, if the memory component of the prototype is represented as the  $s-mem$  component of the state of the abstract machine, and the memory is divided into pages, each page containing a number (presumably fixed) of words, then a reference to a single word will require three functional applications to select the word from the state. To accomplish this will require the provision of two arguments; the word address (or index) and the page address. Thus it would be possible to develop a word reference mechanism in the form of a composite selector function

$s-word(word-address) \cdot s-page(page-address) \cdot s-mem$

Thus the definition of the store operation might be

```

store =
  s-word(s-wa*s-mar( $\xi$ )) * s-page(s-pa*s-mar( $\xi$ )) * s-mem:
  s-mbr( $\xi$ )

```

where  $s-wa$  selects the word address from the memory address register, and correspondingly the  $s-pa$  function selects the page address, and  $s-mbr(\xi)$  represents the memory buffer register into which (by some previous step) the value which is to be stored has been placed.

This complexity of structure is defined in terms of predicates which describe the abstract syntax (i.e., structure) of the state of the machine. In part, for this mythical machine which we have been considering, the state can be defined by the predicates:

```

is- $\xi$  = (<s-mem:is-memory>,
  <s-mbr:is-word>,
  <s-acc:(<s-link:is-bit>,
    <s-body:is-word>)>,
  <s-mar:(<s-ma:is-word-address>,
    <s-pa:is-page-address>)>,>,
  ...)

```

where each of the pairs in the structured predicate specify the name of the branch on which the component is located (in the tree descriptive sense) and the structure of the component. Each of these descriptions must eventually be defined in terms of the elementary objects in the system, so that, for example, the  $s-link \cdot s-acc$  component of the state is defined to be in conformance with the predicate  $is-bit$ , which defines a set of elementary objects. On the other hand, the memory buffer register ( $s-mbr$  component) is defined to be of the form  $is-word$  which we will define by the structure

```

is-word = ({<bit(i):is-bit> | 0 ≤ i ≤ 11})

```

That is, the structure is composed of a set of pairs, the object of each of which is a bit (defined by  $is-bit$ ) and the selector of which is of the form  $bit(i)$  where the value of the index  $i$  is in the range  $\{0,11\}$ . Effectively this defines a 12 bit word.

#### THE BLUE MACHINE

For the purposes of discussion here, let us examine the structure and description of a simple processor. The machine chosen is that described by Foster (Fol) since his description (from a pedagogical point of view) fits our purposes well.

BLUE is a binary, two's complement, stored program, fixed word length, parallel, digital computer with 4096 words of 1  $\mu$ sec co-ordinate addressed core storage of 16 bits per word. Each word may contain

either a 15-bit integer numeric representation plus sign, or a 16-bit instruction composed of a 4-bit operation code and a 12-bit address. No index registers, no indirect addressing and no interrupt facilities are included, though as may be seen from the descriptions, it would not be conceptually difficult to add these features. The general picture of BLUE is shown in Figure 2 and the corresponding representation of the components in the state of the abstract defining machine is shown in Figure 3. For the purposes of our discussion here we shall assume that the external operations of loading the program counter and starting the operation of BLUE by the pressing of the appropriate buttons result in the deposition of the low order contents of the switch register into the program counter and the setting of the run flip-flop to RUN (represented by 1) respectively. No specific descriptions of these actions will be included since these are manual rather than automatic operations. Foster describes the basic cycles of the BLUE machine as being composed of two parts; the FETCH and the EXECUTE cycles. It is assumed that the STATE flip-flop which defines which cycle is to be entered next, will be set to F initially, thereby assuring the correct sequence of operations. The actions of the FETCH cycle are described in Table 1 (from Fol).

TABLE 1

The Fetch Cycle Elements

Clock Pulse	Action	
1	initiate read-restore	} Read time
2	+1 → PC	
3	clear MBR	
4	clear IR	
5	(MBR) → IR	} Begin decode
6	...	
7	...	} Restore time
8	...	
		} May change contents of MAR

The last three pulse times in this sequence are available for the execution of the various non-memory referencing instructions such as HLT (halt), JMP (jump) or CSA (console switches to accumulator), or for the set up operations necessary for the execution of two cycle instructions.

Close examination of the description of the first part of the FETCH cycle (which is common to all BLUE instructions) shows that there are at least two operations occurring simultaneously during pulse times 2 through 4; that is, the action of fetching the instruction from memory (at a location determined by the contents of program counter) initiated at pulse time 1 is operational through pulse time 4, at which time the contents of the memory location are available in the memory buffer register. Whilst this action is continuing the other actions of incrementing the program counter (time 2), and clearing the MBR and IR are executed in parallel. During times 5 through 8, the memory is being restored and thus additional parallel operations are proceeding during these pulse times. This verbal and tabular description can be converted into a VDL instructional system which is equally expressive:

```

fetch =
  part-2;
  register-set,
  initiate-read

```

where

```

register-set =
  clear-ir;
  clear-mbr;
  inc-pc;
  no-op

```

and

```

initiate-read =
  mem-to-mbr;
  no-op;
  no-op;
  no-op

```

FIGURE 2

THE BLUE MACHINE

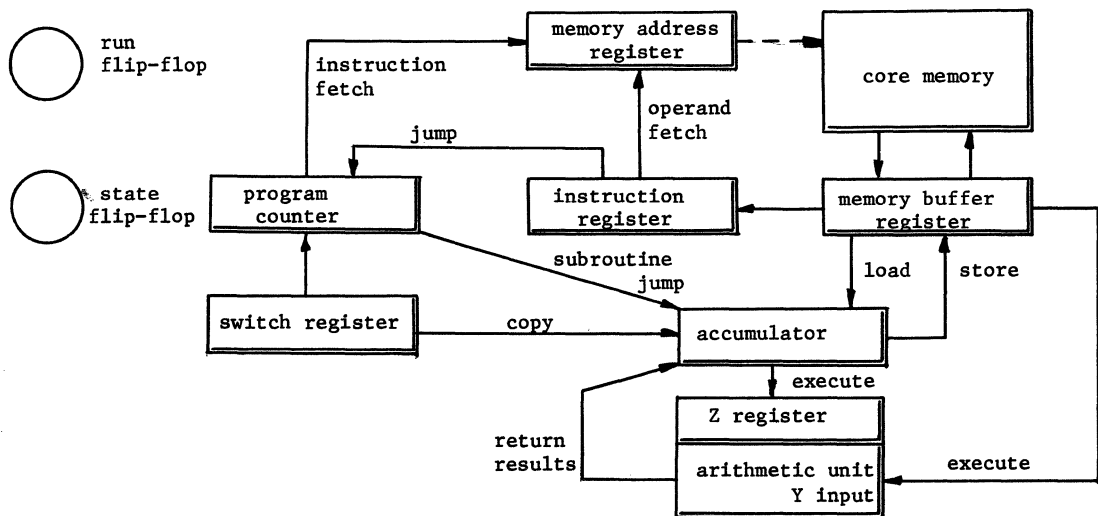
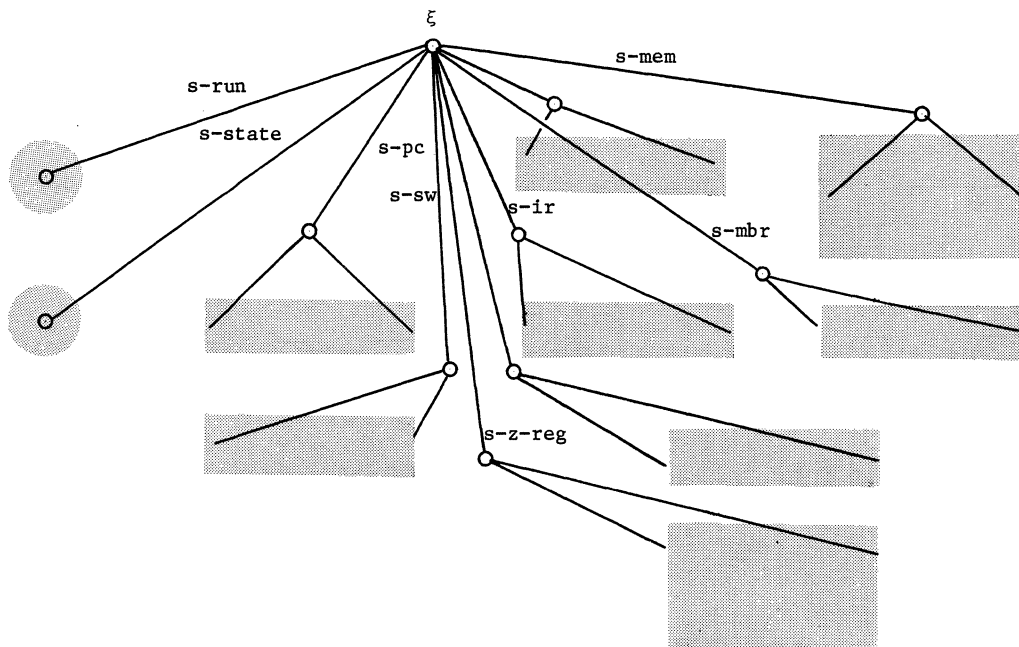


FIGURE 3

THE VDL OBJECT REPRESENTING THE BLUE MACHINE



where the instructions no-op are used to show the relative timing of the two set of instructions. In this case it is not clear from the description of the fetch cycle what actually occurs in the initiate read operation in BLUE during pulse times 1 through 3, though it is clear that in pulse time 4 the contents of the selected location are placed in the MBR. This operation can be defined by the instruction

$$\frac{\text{mem-to-mbr}}{\text{s-mbr:word(s-mar}(\xi)\text{) \cdot s-mem}(\xi)}$$

Similarly, the instruction to increment the program counter may be defined by the group

$$\frac{\text{inc-pc}}{\text{s-pc:s-pc}(\xi) + 1}$$

Immediately we must question whether this definition is sufficient. From the point of view of the programmer, this definition clearly states the action which BLUE is to take; however, from the point of view of the designer (or someone else interested in more details) this definition might better be expressed in the form

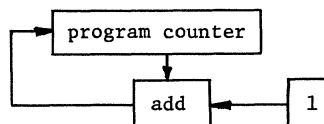
$$\frac{\text{inc-pc}}{\text{s-pc:add}(\text{s-pc}(\xi), 1)}$$

where the function add is to be defined further. For a programmer this depth of definition may well be sufficient, but by considering a function to be equivalent to a logical circuit which could be

defined by a logical expression. In any case this definition can be translated as being representative of the circuit shown in Figure 4.

FIGURE 4

THE PC INCREMENT SYSTEM



Once the two instructions which precede part-2 in the definition of fetch have been cleared off the control stack, then the second portion of the fetch cycle can be initiated. As in the first cycle this contains two parallel actions; the decoding of the instruction and the restoration of the memory. Thus part-2 can be described by the group

$$\frac{\text{part-2}}{\frac{\text{next-state};}{\text{decode}, \text{restore}}}$$

where the decode instruction is expanded into the sequence

$$\frac{\text{sieve};}{\text{mbr-to-ir}}$$

and where sieve is the instruction which replaces itself by the sequence of operations which result in the execution of the BLUE instruction. This instruction can be defined by the conditional expression

```

sieve =
  oct(s-op*s-ir(ξ)) = 0 → execute-hlt
  oct(s-op*s-ir(ξ)) = 1 → execute-add
  ...
  oct(s-op*s-ir(ξ)) = 17 → execute-nop

```

where the selector function (defined in the abstract syntax of BLUE) s-op selects the operation code portion of the instruction from the instruction register (the s-ir component of the state ξ). This portion of the instruction is represented by a tree and therefore true equality can only be attained if the comperand is also a tree. However, we have chosen to overcome this, at this level of definition by the use of the function oct which we define to develop the octal equivalent of the tree representation. This object can then be compared with the octal operation codes. To be more precise at a lower level of definition it would be necessary to describe this sieving operation by logical expressions of the form

```

bit(15)*s-ir(ξ) = 0 &
bit(14)*s-ir(ξ) = 1 &
bit(13)*s-ir(ξ) = 0 &
bit(12)*s-ir(ξ) = 1 → execute-jmp

```

which more precisely mirrors the structure of the binary decoding tree for BLUE.

At the end of the fetch cycle the STATE flip-flop is set to indicate which of the possible two states is to be entered next; E indicates the execute cycle, F indicates the fetch cycle. Thus the instruction next-state which is the final instruction in part-2 is the switch which determines where the processing should continue. An alternative means of specifying the sequence of steps in the fetch cycle which are directly related to the pulse times would be to define the fetch instruction as a sequence of instructions each of which is related to the pulse time and which then leaves the next pulse time operation as the next instruction to be executed. That is,

1. fetch =  
    pulse-time-1
2. pulse-time-1 =  
    pulse-time-2;  
    initiate-read
3. pulse-time-2 =  
    pulse-time-3;  
    inc-pc
4. pulse-time-3 =  
    pulse-time-4;  
    clear-mbr
5. pulse-time-4 =  
    pulse-time-5;  
    clear-ir,  
    mem-mbr

and so on.

This scheme would have the advantage (from the point of view of the reader) that the actions are directly related to the pulse times and the dummy no-op instructions are obviated.

#### SUMMARY

The description and design of BLUE was sufficient to indicate the ability of the VDL techniques for describing the operations of a processor. However this was an exercise in the description of an already existing machine and thus no untoward problems came to light. If VDL were to be used as a design tool then some directions are necessary to derive an implementation from a description. Obviously some simple comparisons can be drawn between instructions and the structure of the machine; that is, for example, state-modifying instructions represent data paths between elements of the machine. Macro-expansion definitions can be interpreted in one of two manners; either an expansion is the passage from one level of description to another, as in the description of the inc-pc instruction, or it represents the sequencing of operations which current state of the machine, as in the case of the instruction decode. The precise manner of discriminating between these two uses is not entirely clear at this time and requires further investigation.

In the version of VDL which is most general, and which has been used for the description of programming languages, the instructions are accompanied by a set of arguments which are passed through the control stack. Such arguments, in a processor, require some medium of transmission and can be construed to be indicative of the need for a register within the prototype. That is, if a definitional instruction cannot be expressed without the use of additional data which is passed through the argument list, then an additional register is required in the prototype together with the appropriate data paths.

This presentation has shown the many levels of description which can be served by a single unified definitional schema and has emphasized earlier that the schema is a continuum from the instruction level of definition to the abstract machine which underlies the system. Work is already in progress to develop the properties of the definitional system (see Lel, ch.2) and to develop means for the validation of definitions.

Finally, it must be recognized that not only has VDL the power to be a definitional system for the description of processors, but also is capable of providing a common base for the definition of other descriptive techniques. This capability may well provide the means by which the equivalence of descriptive elements of other languages can be proved, and further will not require the abandonment of other descriptive techniques merely to satisfy the ambition of a unified approach to processor description.

## REFERENCES

- F01 Foster, C.C., Computer Architecture, Van  
Nostrand Reinhold Pub. Co.,  
New York, NY, 1970, Chapter 5.
- L01 Lee, J.A.N., The Formal Definition of the  
BASIC Language, The Computer  
Journal, Vol. 15, No. 1,  
pp 37-41.
- L01 Lucas, P. & Walk, K., On the Formal  
Description of PL/I,  
Ann. Rev. in Automatic  
Prog., Vol. 6., Pt.3,  
1969, Pergammon Press.
- M01 McCarthy, J., Towards a Mathematical Theory  
of Computation, Proc. IFIP  
Congress 1962, North Holland  
Publ. Co., Amsterdam, 1962.

# A METHODOLOGY FOR PARALLEL PROCESSING DESIGN TRADEOFFS

Charles H. Radoy  
George P. Copeland, Jr.  
G. J. Lipovski  
*University of Florida*

## Abstract

A methodology is developed for determining how much parallelism is optimal if a given job stream is to be executed without multiprogramming. Qualitative design tradeoffs are inferred from the cost-performance effect of parallelism on different hardware subsystems. Measures of software parallelism are analytically related to measures of hardware performance. It is shown that an increase in hardware parallelism may be desirable even though it causes an increase in job processing cost and/or a decrease in hardware efficiency.

## INTRODUCTION

There have been numerous papers written about the impact of LSI on computer architecture. Many authors have pointed out that the technology of the inexpensive computer-on-a-chip will make systems with a high degree of parallel processing and multiprocessing economically feasible (5,6,7,12). Kuck has proposed that, by decomposing a program into its concurrently executable parts, these highly parallel systems will be economically viable even when used to execute one program at a time (monoprogramming) (7). On the other hand, Chen has demonstrated that highly parallel systems are doomed to be very inefficient, and he has suggested that multiprogramming is mandatory if such systems are to be practical (3). This apparent disagreement stimulated the analysis made in this paper. We do not claim to have resolved this conflict in favor of one or the other of these authors. In fact, our inquiry is limited to an analysis of monoprogramming applications. However, we do feel that we have developed a methodology for determining how much parallelism (if any) is optimal if a given job stream is to be executed without use of multiprogramming.

In this methodology we will emphasize the consideration of what the user is willing to pay for a particular computational service. In Section I, we explain how we think this consideration can be applied in the design process. In Section II, we derive certain qualitative design guidelines that can be inferred from this consideration. These guidelines may be obvious to the experienced designer, but we feel that it is significant that they can all be inferred from this one consideration.

The performance of a parallel hardware system will be considerably influenced by the parallelism inherent in the software. In Section III, we present some possible measures of software parallelism and derive expressions relating these measures to measures of hardware performance. In Section IV, we show how one

of these expressions can be used in determining the optimal degree of hardware parallelism.

## Section I

Many different measures of computer performance have been suggested and used. The most common measures used for general purpose computer applications are throughput rate, response time and equipment utilization. Systems designed for less than general purpose use may be evaluated against other measures such as the mean time for high priority jobs to get processed or the mean job starting delay (9). In comparing different hardware equipment, the price of the unit can be included in defining the performance measure, resulting in measures such as price per instruction ratio and price per register ratio (12). Other authors have suggested that the performance measure should not only include cost, but must also include a measure of the effectiveness with which the system provides service to the user (10). We feel that, for the general user, a good measure of the quality of service provided is the time required to process his job. Thus, a computer performance measure should include the cost of processing the job and the time required to do that processing. This is not a new idea. Lehman used these two factors when he suggested that the performance of multiprocessing systems be compared by computing the product of the cost of processing and the job throughput time (8). (Using this measure, the best system would of course have the least product.)

One can, however, argue that Lehman's choice of the product of these two factors is arbitrary; there is no *a priori* reason for selecting the product over any other functional relation between these two quantities. In fact, we claim that a system designer should not work with a simple functional relationship of this sort. The following discussion explains why this is so.

Consider a user who has a particular computational job that he wants done. Assuming that he has some experience in running his job on various systems, he will have a pretty good idea of what he is willing to pay to get the job done. Also, what he is willing to pay will depend somewhat on how long he must wait for his results. From time to time, the job turnaround time that he requires may vary, and as it varies, what he is willing to pay may also vary. Figure 1 illustrates the general way in which the user will relate these two factors. This figure is not meant to be drawn against any scale; it is just meant to illustrate that this curve will have three distinct regions. In Region I, the user is telling us that a further decrease in his job's processing time is of no value to

him, and he will not pay more for this better service. In Region II, he is willing to trade cost for "service" in some manner. In Region III, the processing time is so long that the service is of no practical value to this user.

In Figure 2, points A, B, C, D, E and F represent hypothetical hardware executions of our user's job. They each represent a different system because the same system would always run the job for the same cost with the same processing time. (For simplicity we are not considering systems where interactions with other jobs may influence our job's processing time.) Points A and F represent hardware solutions which are unacceptable to our user. Points B and E are acceptable points, and it is important to note that they are equally acceptable to the user; he does not prefer one of these over the other even though their respective costs and processing times may be markedly different. Points C and D are both preferable to points B and E. (e.g. Since the user is willing to pay "B's" price, he finds "C's" lower price for the same service time preferable.)

FIGURE 1

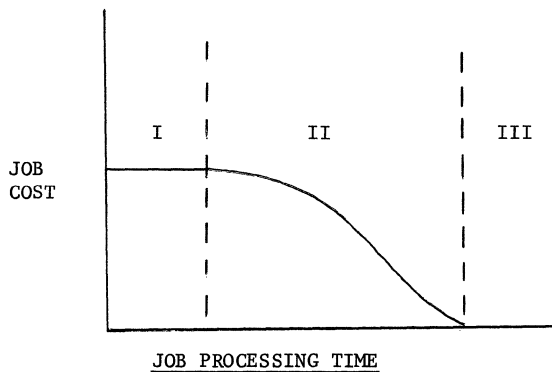
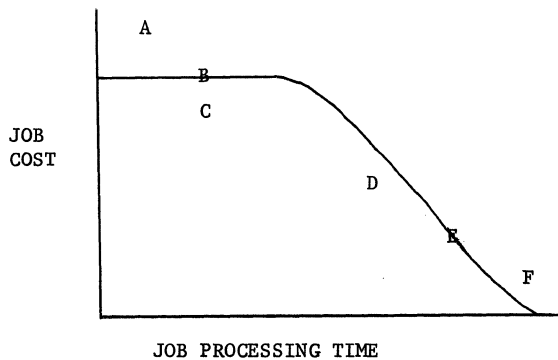


FIGURE 2



Having determined this cost-service tradeoff curve for our particular user, we are unable to say whether or not he would prefer system C to system D. One might suggest that we interrogate our user further concerning his preferences in the region of the graph

below the tradeoff curve. Since we intend that our hypothetical user be representative of a potential market of users, this interrogation would really amount to an extensive market survey. Furthermore, points C and D cannot represent existing systems. Our knowledgeable user would naturally have drawn the curve of what he was willing to pay, in such a way that all existing systems would lie either on or above it. Points C and D can, however, represent designed systems which have not yet been marketed. But, the question of producing system C or D is basically a marketing decision.

The job of the system designer is to produce a design such as system C or D, either of which is clearly better than all existing systems. Thus, the designer needs this curve and a methodology for producing designs which will have "operating points" below it. A valid performance measure could provide this tradeoff curve, but clearly such a measure would not be a simple functional relationship that one could postulate a priori.

So far, we have limited our discussion to the case of one hypothetical user with one job. If we were designing a system for only one user, a design with a projected operating characteristic such as C or D would clearly be a viable project. But a truly viable product would have to provide satisfactory service to many users, and for each user (indeed, for each different job!) there will be a different tradeoff curve.

In order to limit this multiplicity of tradeoff curves, we propose that both the quantities cost-per-job and processing-time-per-job be normalized by dividing them by a measure of the total "work" required by the job. (The quantification of "work" which we propose is discussed in Section III). For instance, if a user has a job that basically consists of two identical subjobs, he will expect the job to cost twice as much and require twice as much time to execute as would one of the subjobs. Since the job contains twice as much work as the subjob, the normalized curves for the job and the subjob will coincide. Furthermore, since the curve is essentially determined by the prevailing market of available computational service, this normalized curve, for a particular type of computation, should not vary appreciably from user to user. Thus, some type of normalization of these curves is required, and we think that this normalization factor is a reasonable one.

Consequently, a curve of this nature can be obtained and it can be of great aid to the designer. For instance, if an existing system has an operating point such as point B in Figure 2 (i.e., it is in Region I of Figure 1), the design of that system can be improved only by a change that will reduce the job processing cost. On the other hand, if one is trying to improve system "E", reducing job processing time is as important as reducing job cost. In the next section we will show how this curve can be used in determining the relative merit of different hardware changes that might be made to an existing design.

## Section II

We will now briefly develop some qualitative hardware design guidelines that can be inferred from the general shape of the user's tradeoff curve discussed in Section I. In this development we will employ the terminology and notation suggested by Bell in categorizing hardware functional modules as data operators (D), controllers (K), etc. (2). We will consider three types of changes that could be made in a design: (1) change of technology used, (2) change of amount of



parallelism in K and (3) change of parallelism in D.

The shape of the curve in Figures 1 and 2 tells us that any design change that both reduces the cost and reduces the processing time will be a good one. (Of course, intuition or common sense could have told us that!) However, if we are not able to reduce both these factors simultaneously, we may still be able to improve the design. If we know the present design results in an operating point in Region I of Figure 1, a design change which reduces the cost of processing will be good even if it increases the processing time. In Region II, a change which reduces processing time while increasing the cost may be desirable.

The use of faster more expensive technology will reduce processing time and may or may not reduce processing cost. Thus, it will in general be a valid design change in Region II, but not in Region I. (In fact, in Region I, the use of slower, less expensive technology will be desirable if it will reduce the cost of processing.)

The use of parallel K (e.g. multiprocessing) will reduce the processing time but will usually not reduce the cost of processing. Thus, increasing the parallelism of K may be a good design change in Region II, while decreasing it may be called for in Region I.

Increasing the parallelism of D while keeping K non-parallel will, up to a point, decrease the cost of processing. The simple example of a parallel adder explains why this is so. As long as the width of the adder can be effectively utilized, doubling the width will halve the add time. But, doubling the width will not double the hardware cost since the cost of the controller will not change. Thus, the processing cost will decrease. At some point, however, due to ineffective use of the increased width, the cost increase will not be offset by the decrease in average add time, and the processing cost will increase. Consequently, with respect to processing cost, there is some optimal degree of parallelism in D. This will also be the optimal degree of D parallelism for a design in Region I of Figure 1. However, in Region II, more parallelism than this "optimal" amount may be desirable.

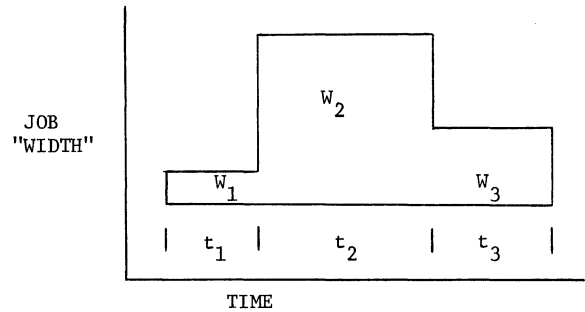
We note that Bell's entire approach of dividing a system into components M, L, K, D, and so on should be useful in analyzing costs. Just as we have evaluated the cost of serial/parallel adders one could evaluate larger systems by the effect of parallelism on each division of the system. The analysis in this section has been qualitative. In the remainder of this paper, we will develop some quantitative relationships which, when used with the user tradeoff curve, can be helpful in determining the optimal degree of hardware parallelism.

### Section III

The optimal degree of hardware parallelism will, of course, be dependent on the parallelism of the job for which it is designed. In discussing job parallelism, we will employ the job "Space-time" diagram suggested by Chen (3). Figure 3 illustrates the space-time diagram of a hypothetical job. The "widths"  $W_i$  represent the relative parallelism of the job during the time interval  $t_i$ . A machine with no parallelism would require  $\sum_i W_i t_i$  time units to process a given job. Thus we define the total work associated with

job to be  $\sum_i W_i t_i$ . This is the factor which we will use to normalize our job-cost versus job-processing-time graph. (Thus, the normalized cost of a job will be  $\text{cost} / \sum_i W_i t_i$ .)

FIGURE 3



An intuitively appealing way to quantify job parallelism would be: parallelism = time-average job "width" or

$$\rho_1 = \frac{\sum_i W_i (t_i / \sum_j t_j)}{\sum_i t_i} = \frac{\sum_i W_i t_i}{\sum_i t_i}$$

Using this measure, the minimum possible parallelism is one, and there is no maximum. This definition of parallelism can be modified so that it takes on values between zero and one by defining

$$\rho_2 = \frac{\sum_i W_i t_i}{\max_i (W_i) \sum_i t_i}$$

In a machine having parallelism equal to  $\max_i (W_i)$ , this definition would correspond to

$$\rho_2 = \text{Space-time used} / \text{Space-time available}$$

Chen has suggested that job parallelism be defined as,

$$\rho_3 = \frac{\text{Amount of space-time showing parallelism}}{\text{total space-time of job}}$$

Letting  $t_s$  be the total time that the job has no parallelism, we have

$$\rho_3 = \frac{\sum_{i \neq s} W_i t_i}{\sum_i W_i t_i}$$

Chen has also defined machine efficiency ( $\eta$ ) to be

$$\eta = \frac{\text{total space-time of job}}{\text{total space-time swept by hardware}}$$

As we increase the parallelism "width" ( $N$ ) of a machine, the normalized processing time for a job,  $T$ , will decrease until  $N = \max_i (W_i)$ . For  $N \geq \max_i (W_i)$ ,  $T$  will have a minimum value.

$$T_{\min} = \frac{\sum_i t_i}{\sum_i W_i t_i}$$

Thus, we see,

$$T_{\min} = 1 / \rho_1$$

For  $T = T_{\min}$ , the maximum efficiency occurs when

$N = \max_i (W_i)$ . If we call this maximum efficiency  $\eta_0$ , we have

$$\eta_0 = \frac{\sum_i W_i t_i}{\max_i (W_i) \sum_i t_i}$$

or  $\eta_0 = \rho_2$

Thus if  $T = T_{\min} = 1/\rho_1$ ,

$$\eta \leq \rho_2$$

Consequently, for highly parallel hardware systems (i.e. where  $N \geq \max_i (W_i)$ ), the software measures  $\rho_1$  and  $\rho_2$  can yield quantitative information about the performance of the system.

The following analysis shows that, if  $N < \max_i (W_i)$ , other quantitative relationships can be derived. If a machine has a parallelism "width" of  $N$ , it will require the interger part of  $(\frac{W_i-1}{N} + 1)$  "passes" to process the  $i$ th parallel section of the job. If we approximate this number of passes to be equal to  $W_i/N$ , our normalized total processing time is,

$$T = (t_s + \frac{1}{N} \sum_{i \neq s} W_i t_i) / \sum_i W_i t_i$$

We note that,

$$T = \frac{1}{N} \rho_3 + \frac{t_s}{\sum_i W_i t_i}$$

Also,

$$(1-\rho_3) = 1 - \sum_{i \neq s} W_i t_i / \sum_i W_i t_i$$

$$(1-\rho_3) = (\sum_i W_i t_i - \sum_{i \neq s} W_i t_i) / \sum_i W_i t_i$$

$$(1-\rho_3) = t_s / \sum_i W_i t_i$$

Thus,

$$T = \frac{1}{N} \rho_3 + (1-\rho_3)$$

We also note that

$$\eta = \frac{\sum_i W_i t_i}{N (t_s + \frac{1}{N} \sum_{i \neq s} W_i t_i)}$$

Thus,

$$\eta = 1/NT = 1/[\rho_3 + N (1-\rho_3)]$$

Consequently, using Chen's parallelism measure, we may easily approximate the normalized job processing time required by a machine having  $N$  levels of parallelism. We now have an analytical means of mapping a job stream containing a range of parallelism into a distribution of normalized processing times. In the next section we will discuss how this will help us determine the optimal degree of hardware parallelism.

#### Section IV

In this section we will illustrate how the user cost-processing time tradeoff curve can be used to determine the optimal degree of hardware parallelism. We will keep design factors such as the technology used fixed and observe the effect on job cost and job processing time caused by varying the degree of hardware parallelism. We will then be able to select the "best" degree of parallelism for a particular job by observing where these points lie with respect to the user's tradeoff curve.

The normalized cost of processing a job is,

$$\text{Cost} = HT$$

where  $H$  is the cost of the system per unit time (rental cost), and  $T$  is the normalized processing time. In parallel hardware systems,  $H$  is of course a function of the amount of parallelism in the system ( $N$ ). If a system is "totally" parallel in the sense that it has  $N$  of all its functional modules (and if the cost of system software is negligible), we might expect the rent associated with this hardware to increase linearly with  $N$ . In that case,

$$\text{Cost} = RNT$$

where  $R$  is the cost per unit time of the basic, non-parallel module.

Many parallel systems are, however, not totally parallel. Parallel processing systems such as the ILLIAC IV and STARAN consist of parallel execution elements under the control of a single instruction decoder (1,11). Doubling the degree of parallelism in such a system does not double its total cost. Also, we can expect that system software costs will not increase in proportion to the degree of hardware parallelism. Consequently, a cost which increases linearly with  $N$  is probably a "worst case" assumption. Perhaps a more realistic assumption would be to use Grosch's Law which states that the system cost will increase in proportion to the square root of the power of the processor. Since the amount of parallelism is a measure of the power of the processor, we have,

$$\text{Cost} = RN^{\frac{1}{2}}T$$

We do not claim that either of these simple formulas is valid for all cases. We will use them merely to demonstrate the methodology which we are developing in this section. Presumably, the designer will be able to fairly accurately estimate the way in which system cost will vary with  $N$  for the particular type of parallelism he is considering. In employing this design methodology, he should of course use his estimate rather than one of these simple formulas.

Figure 4 pertains to the formula

$$\text{Cost} = RN^{\frac{1}{2}}T$$

while Figure 5 illustrates the situation

$$\text{Cost} = RNT$$

In each of these figures, the degree of hardware parallelism is varied from one to eight, and the degree of job parallelism (as measured by Chen's parallelism definition) is varied from 0.5 to 0.95.

Once a designer has obtained a graph of this sort based on his estimates of system cost and job stream parallelism, he can superimpose his user's cost versus

Figure 4

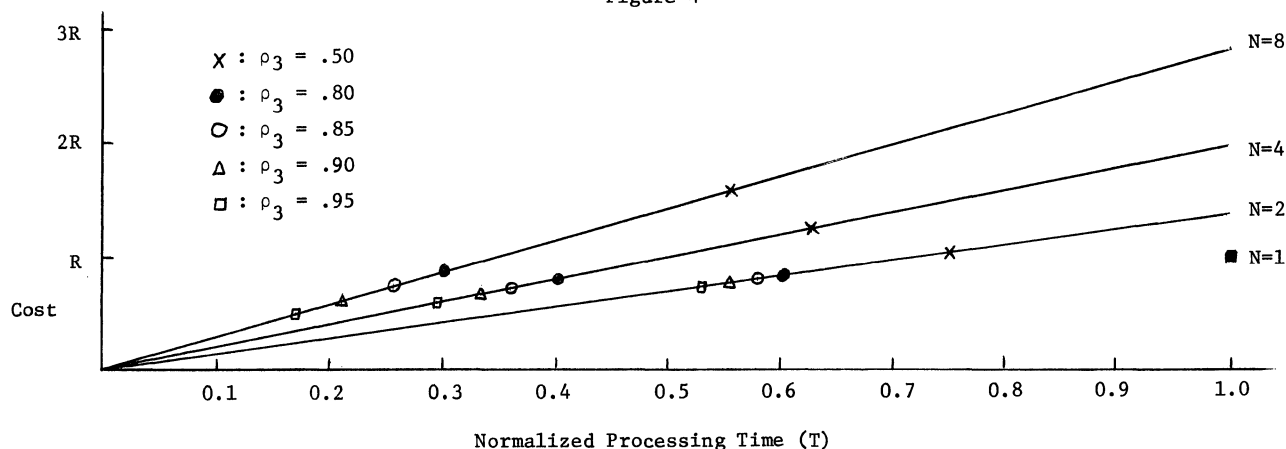
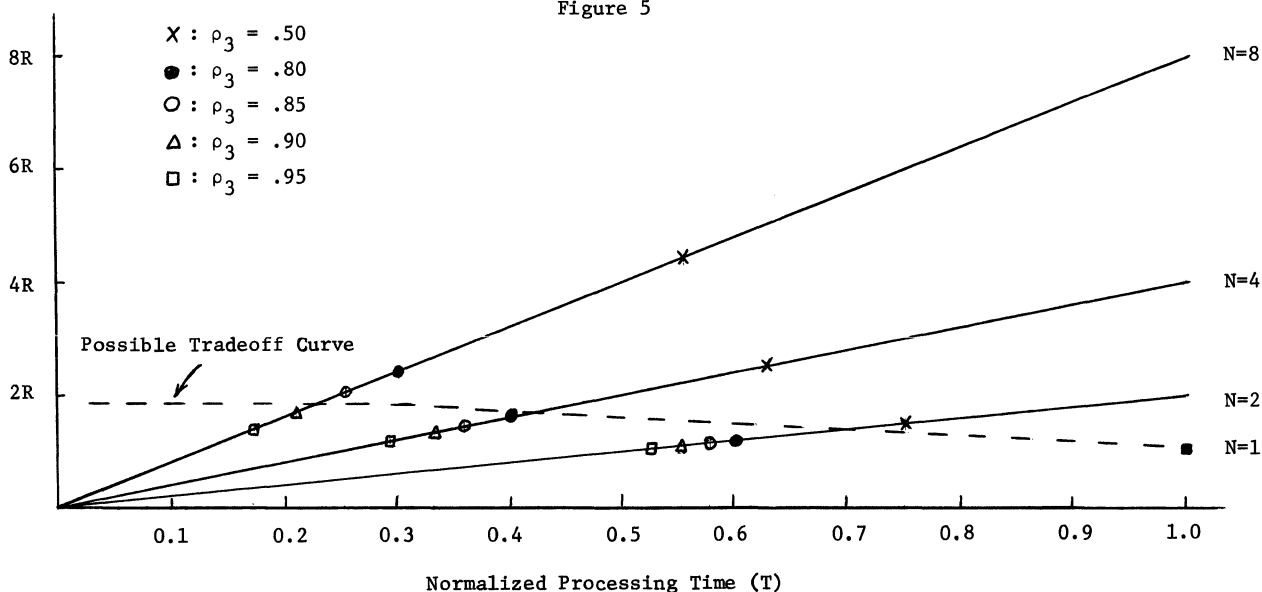


Figure 5



processing time tradeoff curve. Having done this, he can immediately identify which hardware-software parallelism combinations will correspond to viable products. The selection of the "best" of these viable combinations may or may not be trivial.

If we visualize cost-time tradeoff curves of the type illustrated in Figure 1 superimposed on Figures 4 and 5, we can make the following conclusions.

(1) If the non-parallel hardware design produces an "operating point" in Region I of Figure 1, hardware parallelism will be justified only if the software is highly parallel and if cost of the system does not increase linearly with the degree of parallelism.

(2) If the non-parallel hardware design produces an operating point in Region II of Figure 1, some degree of parallelism may be justified even if the software is not highly parallel or even if the system cost increases linearly with the degree of hardware parallelism. (e.g. for the tradeoff curve in Figure 5,  $N = 2$  or  $4$  would be a good design if  $\rho_3 \geq .80$ ).

(3) If the non-parallel hardware design produces an operating point in Region III, some degree of hardware parallelism is mandatory.

(4) As the degree of hardware parallelism is increased, the "spread" of the operating points for a job stream of differing parallelism also increased. Thus, if one has a job stream encompassing a substantial spread of software parallelism, it might be desirable to divide it into subsets having small parallelism variation and then determine the best degree of hardware parallelism for each subset.

As a final point, we wish to make some observations relative to the issue of hardware efficiency. In Section III, we derived the relationship

$$\eta = 1/[N(1-\rho_3) + \rho_3]$$

As Chen points out, this efficiency measure drops rapidly with increasing  $N$ , even if  $\rho_3$  is high. For instance, if  $N = 8$  and  $\rho_3 = .8$ ,  $\eta = .42$ . One would think that a system that was only 42% efficient would

be a poor design and that this combination of  $N = 8$  and  $\rho_3 = .8$  could be rejected on that basis. However, Figure 4 illustrates that, using the design methodology outlined in this paper, this inefficient design might be the best system from the user's point of view. Thus, we feel that even though Chen's definition of efficiency is reasonable, one should not use it as a performance measure in determining if a design is viable. (Of course, we have restricted our investigation to monoprogramming systems. Therefore we do not claim that this comment is necessarily applicable to multiprogramming systems for which high efficiency is a dominant design goal.)

#### SUMMARY

The consideration of the user's cost-performance tradeoff curve has enabled us to present a unified approach to the derivation of important architectural design guidelines. We have derived relationships between "software parallelism" and the performance of systems with different degrees of hardware parallelism. Using these relationships, we have shown that situations may arise where an increase in hardware parallelism is desirable even though it causes an increase in the job processing cost. Also, we have shown that, for a non-multiprogrammed system, the optimal system may exhibit a rather low hardware efficiency.

#### REFERENCES

1. Barnes, G., et.al., "The ILLIAC IV Computer," IEEE Trans. Comput., Vol. C-17, pp. 746-757, Aug. 68.
2. Bell, G., Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.
3. Chen, T., "Parallelism, Pipelining and Computer Efficiency," Computer Design, pp. 69-74, Jan. 1971.
4. Chen, T., "Unconventional Superspeed Computer Systems," Proc. SJCC 72, Vol. 38, pp. 365-371.
5. Chen, T., "Distributed Intelligence for User-oriented Computing," Proc. FJCC 72, Vol. 41, pp. 1049-1056.
6. Foster, C., "A View of Computer Architecture," Com. ACM, Vol. 15, pp. 557-565, July 1972.
7. Kuck, D., "Supercomputers for Ordinary Users," Proc. FJCC 72, Vol. 41, Part I, pp. 213-220.
8. Lehman, M., "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proc. IEEE, Vol. 54, pp. 1889-1901, Dec. 66.
9. Mallach, E., "Job-Mix Modeling and System Analysis of an Aerospace Multiprocessor," IEEE Trans. Comput., Vol. C-21, pp. 446-454, May 72.
10. Rothenberg, D., "An Efficiency Model and A Performance Function for an Information Retrieval System," Information Storage Retrieval, Vol. 5, pp. 109-122, Oct. 69.
11. Rudolph, J., "A Production Implementation of an Associative Array Processor - STARAN," Proc. FJCC 72, Vol. 41, Part I, pp. 229-241.
12. Schultz, G., Holt, R., McFarland, H., "A Guide to Using LSI Microprocessors," Computer, June 73, pp. 13-19.

# DAP—A DISTRIBUTED ARRAY PROCESSOR

Dr. S. F. Reddaway  
Language and Processor Department  
Research and Advanced Development Centre  
International Computers Limited

## ABSTRACT

An array of very simple processing elements is described each with a local semiconductor store. The array may also be used as main storage.

Bit-organisation gives great flexibility, including the minimisation of word length. Use of MSI and LSI is helped by the simplicity of the serial design. Using 15-bit fixed point, the theoretical performance of a 72 x 128 array is about  $10^8$  multiplications or  $10^9$  additions per second. Comparisons are made with other architectures.

Meteorology is considered as an application. It is attractive to have the whole problem in the array storage.

## 1. INTRODUCTION

This paper describes a design study of an array of elements that can be used either as a "Single-Instruction, Multiple-Data stream" (SIMD) processor or as a store. Architectural features of interest are: (a) the use of serial arithmetic to simplify processor logic and optimise store utilisation; (b) an attempt to avoid I/O bottlenecks by mapping complete problems into the array, without relying on overlay techniques; (c) provision for using all or part of the array as a store when not performing its specialised processing functions; (d) the close integration of storage and logic.

The main attractions of array-type SIMD structures are: (a) high absolute performance on certain problems of importance; (b) high performance/cost, partly resulting from using common control logic.

Several examples of this type of architecture have been proposed (1-8) and applications have been suggested in, for example, meteorology, plasma physics and linear programming. Most structures have a single control unit that broadcasts instructions to a regular array of processing elements (PEs) each with individual storage and an arithmetic unit (AU).

Flynn (2) points out four factors that degrade the performance from the theoretical figure given by "Number of PEs times PE performance": (a) Each PE has direct access only to a limited region of store, and excess time may be taken accessing other regions; (b) Mapping the problem onto the array may leave some PEs unused; (c) Owing to overheads in preparing instructions for the array, there may be times when the whole array is idle; (d) While dealing with singularities or boundary conditions the majority of PEs are idle.

These factors are acknowledged to reduce the applicability of such an array. In the present design attempts have been made to mitigate their effect, but the over-riding consideration has been to simplify the PE design; this has been done to the extent that the

theoretical performance is very high, in spite of the AU cost being small compared with that of the storage. In effect, therefore, the store is being adapted to an array processing function. This may be contrasted with attempts to adapt the processor to array operations (e.g. CDC STAR).

A dispersed system, i.e. one with many PEs each with local memory, has potential cost and speed advantages deriving from: (a) reduced "cable" delays; (b) reduced address transforming and checking; (c) faster actual access; (d) simplified data routing and priority logic.

A number of potential PE designs of varying parallelism have been considered for building arrays of the same theoretical performance, with the following general results.

The gate count varies with the degree of internal PE parallelism. A purely serial PE has considerable advantages particularly for low precision work.

Serial PEs have fewer connections at all packaging levels.

The extreme simplicity of serial PEs permits the very effective use of batch fabrication and testing techniques and keeps hardware development rapid and cheap. The small number of circuit and board types helps development, production, spares holding and maintenance.

Serial designs have exceptional functional flexibility; very few decisions are built into the hardware. However, fully indexed addressing is expensive.

The design is somewhat similar to SOLOMON 1 (8); the main differences stem from the exploitation of modern technology.

## 2. THE ARRAY

### 2.1 CONFIGURATION

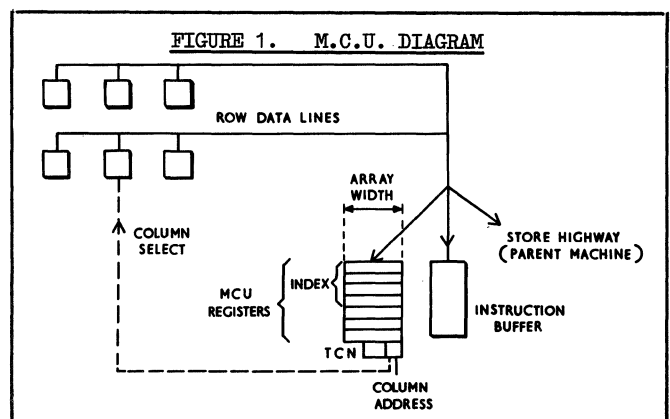


Figure 1 is an overall configuration diagram. The rectangular array has an essentially two dimensional nearest neighbour connectivity, and has one dimension matched to the store highway of a conventional computer (the "parent" machine). This connection provides the route for loading both data and array instructions into the array storage for array processing; it also permits the parent machine to use the array storage as its own main storage. Input/output is done by the parent machine.

The Main Control Unit (MCU) has: (a) a conventional instruction fetching arrangement; (b) an instruction buffer whose purpose will be described later; and (c) a set of registers, many of which can be matched to the array by row or column for a variety of purposes, one of which is indexing. For sizable arrays the MCU is a very small fraction of the total hardware.

After loading, the bits of a word are spread along a column of PEs, and this method of holding data is termed Main Store mode. Another method, termed Array mode, stores all the bits of a word in a single PE. This is more attractive for processing large arrays, but requires initial and final transformation of the data from and to Main Store mode; this is done inside the array.

## 2.2 THE PE

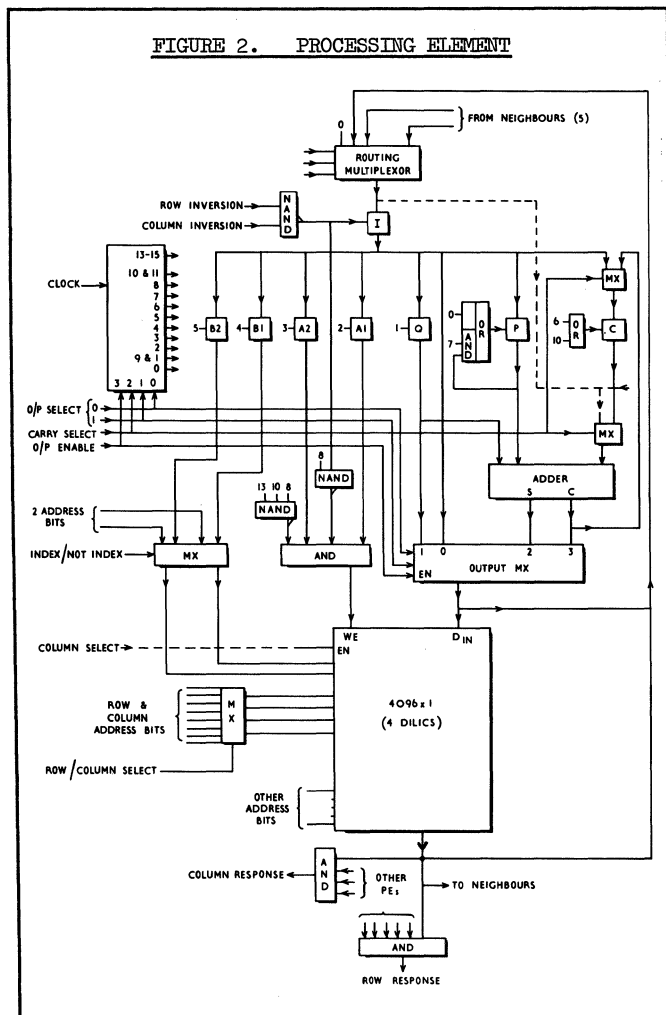


Figure 2 is a PE diagram. The registers are all one-bit; P and Q are for operands, C is the carry register, A1 and A2 are activity bits that can prevent writing to

store, and B1 and B2 can supply 2 address bits. The routing multiplexor can select a bit from the PE's own store, or from a neighbour's store, for writing to a register; selecting zero and controlling its inversion permits data input from outside the array (for example, an MCU register). The sum, carry, data input or contents of Q can be output from the logic, usually to the store. The store contents can be output externally (to, for example, an MCU register) via the gates at the bottom of Figure 2; the bits output can be either from a selected column of PEs, or the logical AND of rows (or columns) of PEs. One use for the latter is for a test over all PEs.

The fifth "neighbour" connection is to the PE half a row away in the same row; this permits both faster mass movement of data around the array, and a "2 $\frac{1}{2}$ D" PE geometry. Bit patterns in one or two MCU registers can be applied to the "inversion" inputs to produce a veto selective by rows and/or columns on writing to PE stores. Figure 2 shows 4 address bits capable of being selected by row or column; what indexing facilities should be provided is still an area of debate.

Some differences from the PE in (7) are: (a) more row/column symmetry; (b) a latch feature (shown on the P register) for associative comparisons; (c) data can be shifted directly between PEs without using the store; (d) input data can be loaded directly into store; (e) there is a ripple carry path between PEs for Main Store mode arithmetic; (f) the bipolar store is now 4K instead of 2K.

It is intended to package 2 PEs minus their stores and routing multiplexors in one 24 pin integrated circuit.

## 2.3 EDGE CONNECTIONS

For instructions that involve neighbours, it is the array geometry that determines what happens at the array edges. Rows or columns may be: (a) cyclic, with their ends connected together; (b) linear, with a continuation onto a neighbouring line; (c) as (b) but with the extreme ends connected; or (d) plane, with external data applied at the relevant edge. In addition, a row may be considered in two halves (2 $\frac{1}{2}$ D geometry). There are thus 32 geometries, and they are set by program.

## 2.4 CONSTRUCTION

A board would contain a 6 x 4 PE section with 4K bits/PE; there would be 137 external connections and 173 ICs, 96 of them for storage. The array can be viewed as doing processing in the store, and costs only about 25% more than ordinary storage made out of the same technology. A platter would contain a 36 x 16 PE section; the number 36, and multiples of it, match standard store highways. "Folding" of the array makes connections between the extreme edges short.

The economy obtained by the dense packing of the integrated circuits is the result of the favourable marriage of space-limited (or power-dissipation limited) storage and pin-limited logic.

## 2.5 TIMING

Because most micro-instructions do not involve a response from the array, the equalisation, rather than minimisation, of delays is important. Even with a comparatively slow logic technology, the micro-instruction rate should be about 5-6 MHz; the storage

element delays are the biggest factor, and this illustrates how the array can exploit bipolar store speeds, unlike a large conventional machine.

## 2.6 FUNCTIONS

In (7) the basis of the micro-programming notation is given and it is shown how Array mode fixed and floating point instructions are built-up. Bit organisation means that only necessary work need be done; for example, multiplication only needs to calculate a single length result.

Code for execution must be compiled down to the one-bit micro-instructions, except that for working regularly along the bits of words a short loop can be constructed. This loop is held in the instruction buffer, so that no further instruction fetching from the array storage is needed during execution of the loop. This feature reduces the instruction fetching overhead from 100% to about 20%. Subroutine construction will be possible.

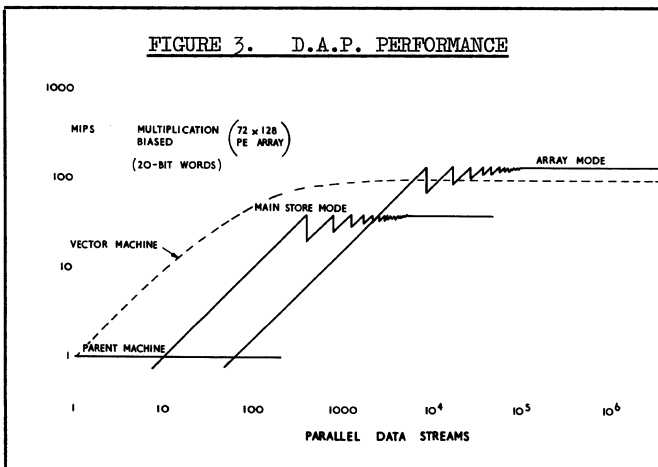
## 2.7 PERFORMANCE

For array mode, fractional fixed point multiplication takes about

$$\frac{n(3n+13)}{2}$$

micro-instructions where  $n$  is the word length; fixed point addition takes little more than  $3n$  micro-instructions. Floating point takes a little longer for multiplication, and considerably longer for addition (see (7)). 20-bit multiplication takes about 730 micro-instructions plus about 160 cycles for micro-instruction fetching, and at  $5\frac{1}{2}$  MHz would take about 160  $\mu$ sec; 20-bit addition takes about 12  $\mu$ sec. Multiplication of an array by a common number can be about four times faster.

Main store mode arithmetic is faster than Array mode for smaller arrays. In terms of absolute speed, addition is about 11 times faster and multiplication, using a carry save technique ending with a ripple carry, is about six times faster for 20 bit precision (the latter factor increases with the precision).



The user has three modes of working at his disposal: the parent machine for scalar working, Main Store mode for small arrays and Array mode for large arrays. Figure 3 shows roughly what is possible in the three modes; the useful processing rate in Million Instructions (or, more accurately, results) Per Second (MIPS)

is plotted against the number of parallel data streams for the type of computing indicated and a 9200 PE array. Only the top ends of the sloping lines depend on array size. The dashed line shows the similar graph for a powerful vector machine (there are many other differences between the two types of machine).

The overall performance depends on the application and programmer skill.

## 2.8 A COMPARISON

ILLIAC IV is a well known machine, so a brief comparison is attempted with Array mode, assuming the problem parallelism is sufficient to occupy either machine. Many differences are not easily quantifiable, but as a starting point the main assumptions for a numerical comparison are given in Figure 4. The first four lines give the instruction mix;  $B$  is the number of bits precision for the serial design, which has no separate store accesses because all functions are store-to-store.  $P$  is the clock period (180 nsec). 20% is subtracted from the ILLIAC IV totals to allow for instruction overlap.

**FIGURE 4. DESIGN COMPARISON**  
**ILLIAC IV ASSUMPTIONS**

INSTRUCTION MIX AND TIMINGS:

	SERIAL DESIGN	ILLIAC IV			
		SINGLE PRECISION	DOUBLE PRECISION	TRIPLE PRECISION	
1 ADD/SUBTRACT	$(2+3B)P$	0.125	0.25	0.5?	$\mu$ sec
1 MULTIPLY	$(4B+1.5B^2)P$	0.25	0.5	2.0?	$\mu$ sec
2 STORE ACCESSSES	0	0.325	0.65	1.0?	$\mu$ sec
1 MODE SETTING (etc.)	$4P$	0.05	0.05	0.05	$\mu$ sec
TOTAL	$(6+7B+1.5B^2)P$	0.75	1.45	3.55?	$\mu$ sec
TOTAL	-20%	0.6	1.16	2.8?	$\mu$ sec
MANTISSA		25	49	73	BITS
EXPONENT		7	15	(23)	BITS
"USEFUL" EXPONENT		4	6	8	BITS

LOGIC/PE.  
ILLIAC IV ~12000 FAST ECL GATES  
SERIAL DESIGN ~60 TTL GATES  
1 FAST ECL = 2TTL GATES  
RATIO = 200 x 2 = 400

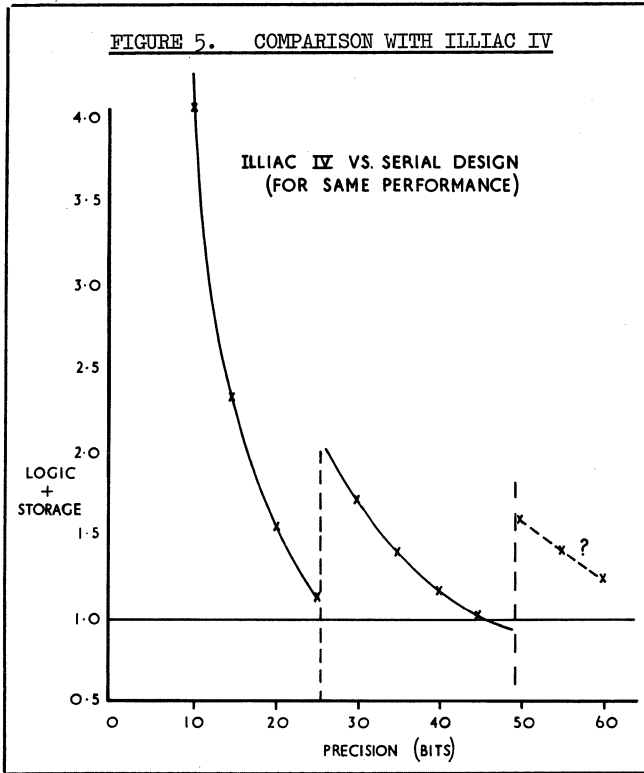
Figure 5 compares the hardware required to build an array of given performance for words of a particular precision. Logic and storage have equal weight; Figure 4 gives the gates/PE ratio and the storage comparison involves an estimate of the unnecessary bits in the ILLIAC IV word. The graph would favour ILLIAC IV only for working exclusively with 46-49 bit precision. At low precisions serial PEs have a very big advantage.

Such numerical comparisons are of only limited value. For example, the vertical scale of Figure 5 would be multiplied by about 4 if integrated circuit count were used as a hardware measure. Other factors such as hardware simplicity and repetition, pin counts and functional flexibility are equally important.

## 2.9 EXAMPLE OF STORAGE ECONOMY

For problems with large amounts of data, storage economy is important, particularly if it permits storing the complete problem in the array. The user can apply various tricks. As an example, consider three dimensional field problems. In order to prevent physical "truncation" errors, programs are designed so

FIGURE 5. COMPARISON WITH ILLIAC IV



that differences between neighbouring variables require fewer significant bits than the variables themselves. If variables have to be held simultaneously for two time steps, then, for example, they can be grouped into sets of 16 nearest neighbours in space and time ( $2 \times 2 \times 2 \times 2$ ), and held as follows: (a) a short floating point number close to the maximum of the group (maybe a 4-bit mantissa and 3-bit exponent); and (b) 16 differences in block floating point (maybe 12-bit mantissas and a common 2-bit block exponent). This results in 12.6 bits/variable and is roughly equivalent to floating point with a 15-bit mantissa and 3-bit exponent, i.e. a gain of nearly 50%; other machines require floating point variables to occupy up to 64 bits, i.e. up to 5 times more.

3. METEOROLOGY AS AN APPLICATION

This is considered more fully in (7). Meteorology includes both simulation experiments and forecasting, and as simulation programs are central to both, attention will be confined to them. (Forecasting also uses analysis and initialisation programs to assimilate the "real" data). For simulation programs, the frequency of add/subtract and multiply instructions is roughly equal, and divide is much less frequent. For DAP, multiplication takes much longer than addition, so the number of multiplications and their timing give a first approximation to the speed of a program.

The table gives a rough guide to parameters in use today and those that should be aimed at.

Using the 18 bit (fixed point) precision suggested in Section 3.3, each PE can perform a multiplication in about 140  $\mu$ sec. Section 3.2 discusses the efficiency of PE usage; 50% might be a reasonable figure. Thus about 8000 PEs are adequate to perform the  $2.5 \times 10^7$  multiplications per second indicated above.

TABLE

	Present		Next stage
	Forecast Programs	Global Research Programs	
Number of Vertical Columns of Grid Points	3000	10 000	x 4
Number of vertical levels	10	5	x 2
Total number of variables	$2 \times 10^5$	$2.1 \times 10^5$	x8 ( $1.6 \times 10^6$ )
Time step	2 min.	5 min.	$\div 2$
Number of time steps	1000	10 000	x 3
Multiplications per column per time step	1000	500	x 2.5
Multiplications/sec.	$1.2 \times 10^6$	$1.2 \times 10^6$	x20 ( $2.5 \times 10^7$ )
Speed-up over real time	50-100	50-100	50-100

3.1 STORAGE

It may be tempting to use a backing store for big problems; however, the smaller the array storage the larger is the channel capacity required. In (7) an example was studied of a problem using explicit integration which had  $1.5 \times 10^6$  variables of average length 20 bits, and was processed on an 8200 PE array with an I/O channel of  $10^7$  bits/sec. Three formulations of the problem had the following trade-offs: (a) 1850 bits/PE and speed degraded by a factor of 2.5, (b) 2800 bits/PE and speed degraded by 1.3, and (c) 4600 bits/PE, the complete problem in the array and no degradation. A similar problem using implicit methods would have its speed degraded by an order of magnitude if a backing store was used.

This sort of problem needs about  $5-10 \times 10^7$  bits of storage. The falling cost of semi-conductor storage makes this amount of array storage feasible, and the simplicity and reliability of a unified semi-conductor system makes it attractive. Partly for these reasons, the array has more resources devoted to storage than to logic.

3.2 PARALLELISM

Efficiency, defined as the fraction of time a PE is active, depends on programmer skill as well as the problem. Numerical procedures used at present have usually been devised with serial machines in mind, and sometimes a slightly different procedure may be much more efficient.

Explicit methods for the "basic" meteorological equations are efficient. Boundaries do not have much effect because it is usually a case of omitting things. "Secondary" effects may cause efficiency to drop. The computation is different if the air is saturated.



Convection may require the checking of neighbouring vertical layers for stability, followed by a relaxation process. Study indicates that these effects need not have a major effect on the overall efficiency.

Once various conditions have been established "branching" by means of activity bits is very rapid, and can be done frequently in order to improve parallelism. (A conditional branch in a conventional program loop, or selection in a vector machine, are slow by comparison).

Implicit methods involve either ADI (alternating direction implicit) or relaxation methods; the former are not particularly efficient but the latter are.

There seem to be 4 types of grid in use: (a) rectangular for fairly local forecasts; (b) octagonal in overall shape (rectangular neighbour connection) for the northern hemisphere; (c) cylindrical on a global latitude-longitude basis; (d) as (c) except that the number of points on a line of latitude is reduced as the poles are approached. (a) and (c) can fit a rectangular PE array. (b) and (d) would waste some of the PEs. (c) has reduced efficiency because a smoothing process is applied more times near the poles; this can be viewed as a trade-off for the wasted PEs of (d).

### 3.3 PRECISION AND NUMBER REPRESENTATION

Precision costs time and storage space, so that big problems should use only the minimum consistent with accumulated round-off error being small compared with other errors. Different variables can use different number representations and precisions. Knowledge of requirements is only patchy, but should improve; the pay-off, compared with fairly cautious starting schemes, might be a factor of about 1.5 in storage and 2 in speed.

Meteorology is largely concerned with absolute rather than relative accuracy, and the maximum possible values of variables are well understood; this points to either fractional fixed point or a simple floating point. Block-floating of arrays (9) can also be implemented efficiently.

An example of possible economy in space and speed occurs in explicit integration schemes; the increments to variables require considerably less precision than the full variables.

Careful choice of rounding method in order to avoid bias can also lead to economy (7).

A reasonable estimate of the average precision required for fractional fixed point variables might be 18 bits and rather less for the mantissa of floating point variables.

### 4. OTHER APPLICATIONS

An algorithm to solve the two dimensional Poisson's equation was studied. It used a Fast Fourier Transform technique, but the extensive data shuffling that this involved occupied only 20-25% of the time. There was also reduced parallelism in places, and a typical PE was idle about 50% of the time. On a 72 x 64 PE array, a 256 x 256 mesh was estimated to take 50 msec for 20-bit numbers; this compares very favourably with conventional machines. An interesting aspect is that the main array is held in Array mode and certain row and column features are dealt with in Main Store mode; Main Store mode vectors are combined with the array elements in single arithmetic operations.

For the array to be useful, problems must fulfil three conditions: (a) Processing, as opposed to I/O, must be important; (b) Much of the problem must be programmed with parallel and identical operations (these may, however, be selective); (c) Excessive time should not be spent shuffling data round the array. (In some cases this means the data should be fairly regular).

These requirements are not very severe, and the biggest barrier to widespread use is likely to be in devising an acceptable programming language. (In spite of many problems being naturally parallel, many users are indoctrinated by sequential thinking).

Some applications for array processors are discussed in (5). Further applications are suggested by the fact that the array can be used as an "associative processor"; examples might be air traffic control, graphics processing and symbol processing. Associative information retrieval can look attractive over quite a wide range of parameters; with the associative latch, each PE can scan 1 bit every micro-instruction, and so 10 000 PEs can scan  $5 \times 10^{10}$  bits/second.

The user has the freedom to optimise and experiment from the bit level upwards; this may help him understand his real computing requirements. The array is not arithmetic biased, and the functional flexibility permits functions to be tailored for all sorts of purposes. The hardware simplicity permits parameters such as the number of bits/PE and the type of storage to be varied easily; for example, a slower, cheaper MOS version would extend the range of applications considerably. The array modularity (almost like storage modularity) means that sizes from 500 to 30 000 PEs are reasonable.

### ACKNOWLEDGEMENTS

The author would like to thank the Directors of ICL for permission to publish and J.K. Iliffe for his support and for originating many of the ideas. The contribution of A.W. Walton is also gratefully acknowledged.

### REFERENCES

1. Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., and Stokes, R.A. "The ILLIAC IV Computer", IEEE Transaction on Computers, C-17, p. 746 (1968).
2. Flynn, M.J., "Some Computer Organisations and their Effectiveness", IEEE Transactions on Computers, C-21, p. 948 (1972).
3. Goodyear Aerospace "STARAN - A New Way of Thinking". A Goodyear Aerospace brochure, Akron, Ohio (1971).
4. Huttenhoff, J.H., and Shively, R.R. "Arithmetic Unit of a Computing Element in a Global, Highly Parallel Computer", IEEE Transactions on Computers, C-18, p. 695 (1969).
5. Kuck, D.J. "ILLIAC IV Software and Application Programming", IEEE Transactions on Computers, C-17, p. 758 (1968).
6. Murtha, J.C., "Highly Parallel Information Processing Systems" in "Advances in Computers". Vol.7, (1966).
7. Reddaway, S.F., "An Elementary Array with Processing and Storage Capabilities", International Workshop on Computer Architecture, Grenoble, June 1973.
8. Slotnick, D.L., Borck, W.C., and McReynolds, R.C., "The Solomon Computer", Fall Joint Computer Conference 1962, p. 97.
9. Wilkinson, J.H., "Rounding Errors in Algebraic Processes", H.M.S.O. London (1963).



# MAXIMAL RATE PIPELINED SOLUTIONS TO RECURRENCE PROBLEMS

Peter M. Kogge  
IBM Corporation  
Owego, N. Y.

## ABSTRACT

An  $m^{\text{th}}$  order recurrence problem is defined as the computation of  $X_1, \dots, X_N$ , where  $X_i = f(\underline{a}_i, X_{i-1}, \dots, X_{i-m})$  and  $\underline{a}_i$  is a set of parameters. On a pipelined computer, where the total stage delay in computing  $f$  is  $d_f$  time units, the solution output rate is one new  $X_i$  each  $d_f$  time unit. This paper describes a method for increasing this rate to 1 per time unit when the function  $f$  has certain simple functional properties. The total stage delay and complexity of the resulting pipelines are also described.

conditions under which the performance of pipelined solutions to first-order recurrence problems can be increased. Section IV generalizes this to  $m^{\text{th}}$ -order recurrences. In all sections, both total pipeline stage length and pipeline complexity are discussed.

The basic background for this paper originates in a series of earlier papers on the solution of recurrence problems on parallel computers (1, 2, 3).

## I. INTRODUCTION

An  $m^{\text{th}}$  order recurrence problem is defined as the computation of the sequence  $X_1, \dots, X_N$  given only

1. Initial conditions  $X_0, X_{-1}, \dots, X_{1-m}$
2. "parameter vectors"  $\underline{a}_1, \dots, \underline{a}_N$ , where each  $\underline{a}_i$  is a collection of solution-independent parameters
3. a "recurrence function"  $f$ .

such that for each  $i, 1 \leq i \leq N$ ,

$$X_i = f(\underline{a}_i, X_{i-1}, \dots, X_{i-m}) \quad (1)$$

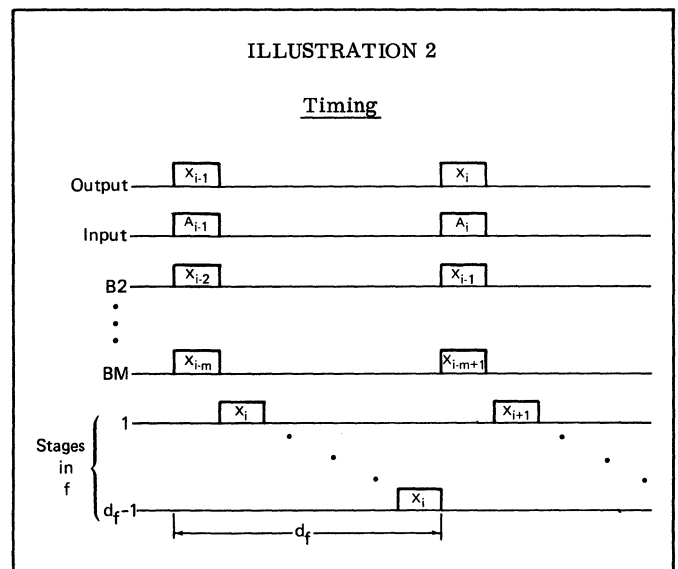
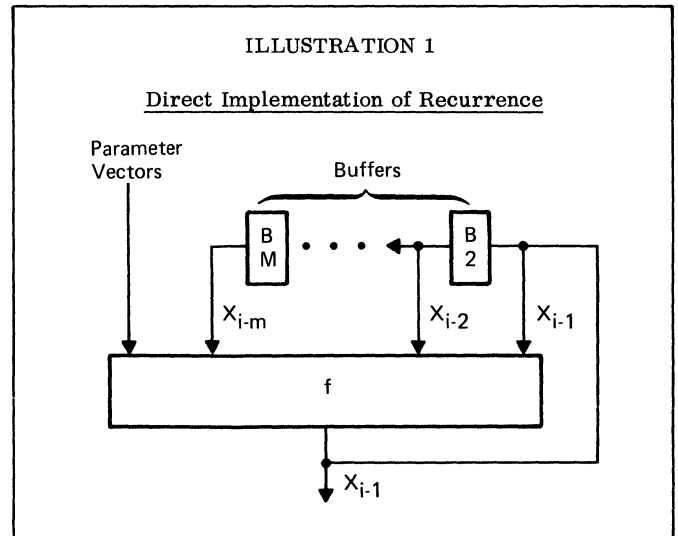
An example is the  $m^{\text{th}}$  order linear recurrence

$$X_i = \sum_{r=1}^m \underline{a}_i(r) X_{i-r} + \underline{a}_i(m+1) \quad (2)$$

A pipelined computing device is one that accepts inputs at a rate of one every  $r$  units of time and produces corresponding outputs  $p$  time units later,  $p \geq r$ . Up to  $\lceil p/r \rceil$  separate computations can be active within the pipeline at one time. For this paper,  $r = 1$ , and thus a pipeline may be considered a series of  $p$  independent "stages," each capable of holding a partial computation on a distinct set of inputs.

Assuming that the function  $f$  is computable by a pipelined device with  $d_f$  stages, a direct solution of a recurrence problem is pictured in Illustration 1. Assuming that  $X_{i-1}$  is output at time  $j$ ,  $X_i$ , which depends on  $X_{i-1}$ , cannot be output until  $X_{i-1}$  has cycled through the entire  $d_f$  stages of the pipeline; i.e., until time  $j + d_f$ . Illustration 2 diagrams the timing of the pipeline. Thus the output rate is at most one element of the sequence per  $d_f$  time units.

The purpose of this paper is to investigate the conditions under which pipelined networks can be configured to have data rates higher than  $1/d_f$ , up to 1 sequence element per time unit. Section II describes a simple example of this procedure. Section III details some



## II. AN EXAMPLE

One of the simplest nontrivial recurrence problems involves a recurrence equation of the form  $X_i = a_i X_{i-1}$ , where  $a_i$  is a real number expressed in floating point notation. The

\*  $\lceil x \rceil$  is the smallest integer not smaller than  $x$ .

function  $f$  in this case is multiplication, a typical implementation of which might involve a two-stage pipe, one stage for exponent addition and one stage for mantissa multiplication. With such an implementation, a direct solution like Illustration 1 would have an output rate of  $1/2$  -- 1 new  $X_i$  every other time unit. The pipelined nature of the multiplier is not exploited.

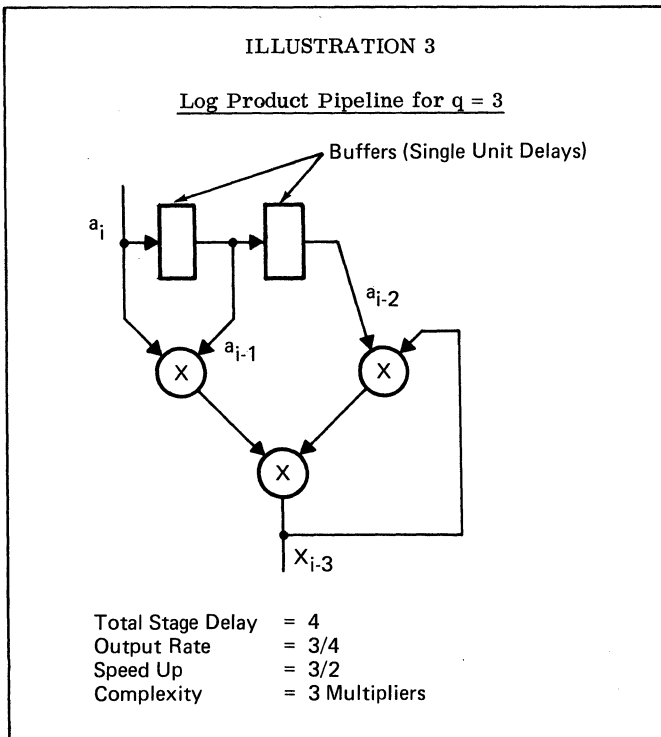
However, the basic recurrence can be rewritten as

$$X_i = a_i a_{i-1} \dots a_{i-q+1} X_{i-q} \quad (3)$$

for any value of  $q$ . Each  $X_i$  in this case requires  $q+1$  numbers to be multiplied. Using the well known "log reduction" technique<sup>(4)</sup>, however, multiple multipliers can be arranged in a tree-like arrangement that computes equation 3, and requires at most  $d(q) = \lceil \log_2 q+1 \rceil$  multiplier delays (compare Illustrations 3 and 4). If  $X_{i-q}$ , for example, is available from such a network at time  $j$ , then  $X_i$  can be computed by time  $j + d(q)$ . This places an upper bound on the output of  $q$  different  $X$ 's ( $X_{i-q+1}, \dots, X_i$ ) in time  $d(q)$  as an output rate of  $q/d(q)$ . This rate is maximized to 1 -- a distinct  $X_i$  in each time unit -- if  $q \geq d(q)$ . For our example, this relation is

$$q \geq 2 \lceil \log_2 q+1 \rceil \quad (4)$$

which occurs for  $q \geq 6$ . Illustration 3 diagrams a log-product pipeline solution of equation 2 for  $q = 3$ ; the output rate is  $3/4$ , a factor of 1.5 better than the direct implementation but still not maximal. Illustration 4 diagrams a maximal flow pipeline for  $q = 6$ .

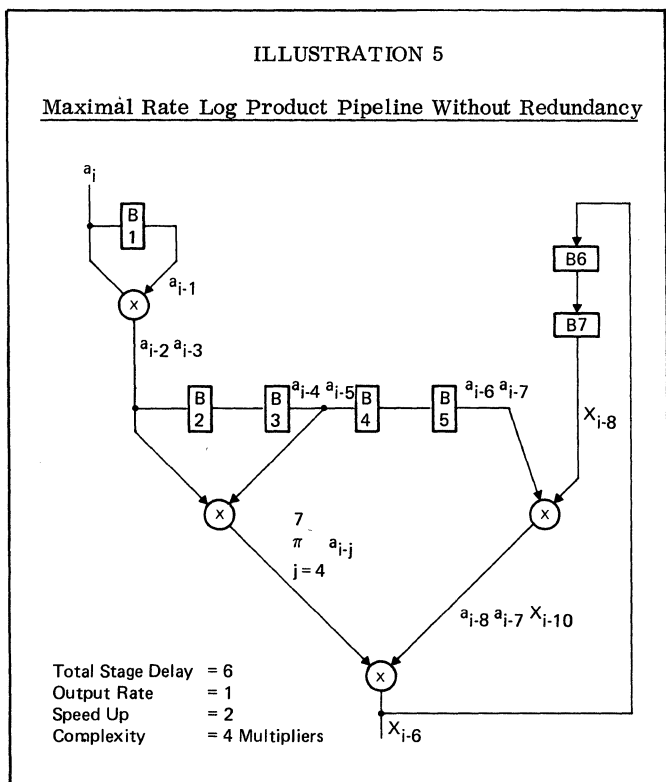
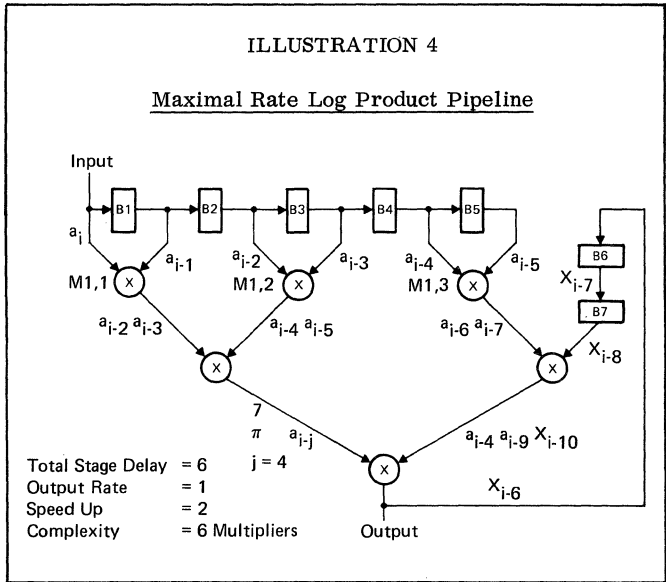


Several comments should be made in respect to the maximal rate pipeline of Illustration 4:

1. Buffering is used to equalize the delays in all sections of the pipeline to 6 time units.

2. At each time unit, all buffers and all stages of all multipliers are computing products that will lead to some element of the solution sequence; i. e., the pipeline is fully loaded.
3. The multipliers labeled M1,2 and M1,3 are redundant in that any calculations they perform were performed earlier by M1,1.

The redundant M1,2 and M1,3 can be removed by moving the buffers B2 - B5 from the inputs to M1,2 and M1,3, and placing them on the output of M1,1, as shown in Illustration 5. This pipeline still exhibits maximal flow, but involves no redundant computations.



### III. FIRST-ORDER RECURRENCE

The key behind the applicability of the log-reduction techniques on the example of the last section was the associativity of the recurrence function multiplication. Although many recurrence functions are associative, and are solvable in a manner identical to that used above, most of the more common recurrences are not. As detailed in earlier papers, however, a large class of problems, particularly first-order problems, have a property similar to associativity that is as useful in configuring maximal rate pipelines<sup>(1, 2, 3)</sup>. This property is termed "semi-associativity," and is defined as follows:

#### DEFINITION 1

A recurrence function,  $f$ , is said to be semi-associative with respect to a companion function,  $g$ , if there exists a function  $g$  such that for all parameter vectors  $\underline{a}$  and  $\underline{b}$  and all  $x$ 's:

$$f(\underline{a}, f(\underline{b}, x)) = f(g(\underline{a}, \underline{b}), x) \quad (5)$$

An easily provided corollary to this definition is that with respect to its effects on  $f$ , the companion function,  $g$ , is associative.

#### Corollary 1

For all parameter vectors  $\underline{a}$ ,  $\underline{b}$ , and  $\underline{c}$ , and all  $x$ :

$$f(g(\underline{a}, g(\underline{b}, \underline{c})), x) = f(g(g(\underline{a}, \underline{b}), \underline{c}), x) \quad (6)$$

Examples of recurrences that have a companion function are:

$$x_i = \underline{a}_i(1) x_{i-1} + \underline{a}_i(2) \quad (7)$$

$$x_i = \underline{a}_i(2) x_{i-1} \underline{a}_i(1) \quad (8)$$

$$x_i = \frac{\underline{a}_i(1) x_{i-1} + \underline{a}_i(2)}{\underline{a}_i(3) x_{i-1} + \underline{a}_i(4)} \quad (9)$$

In the following descriptions, it is assumed that pipelined computing modules can be built for both  $f$  and  $g$ , and the number of inherent stage delays is  $d_f$  and  $d_g$ , respectively.

The existence of a companion function allows a first-order recurrence

$$x_i = f(\underline{a}_i, x_{i-1}) \quad (10)$$

to be placed in the following form for any  $q$

$$x_i = f(g(\dots g(\underline{a}_i, \underline{a}_{i-1}), \dots \underline{a}_{i-q+1}), x_{i-q}) \quad (11)$$

The associativity of  $g$  with respect to  $f$  allows a log-reduction network to compute the  $g$  composition portion of equation 11 in  $\lceil \log_2 q \rceil$   $g$  computation delays.

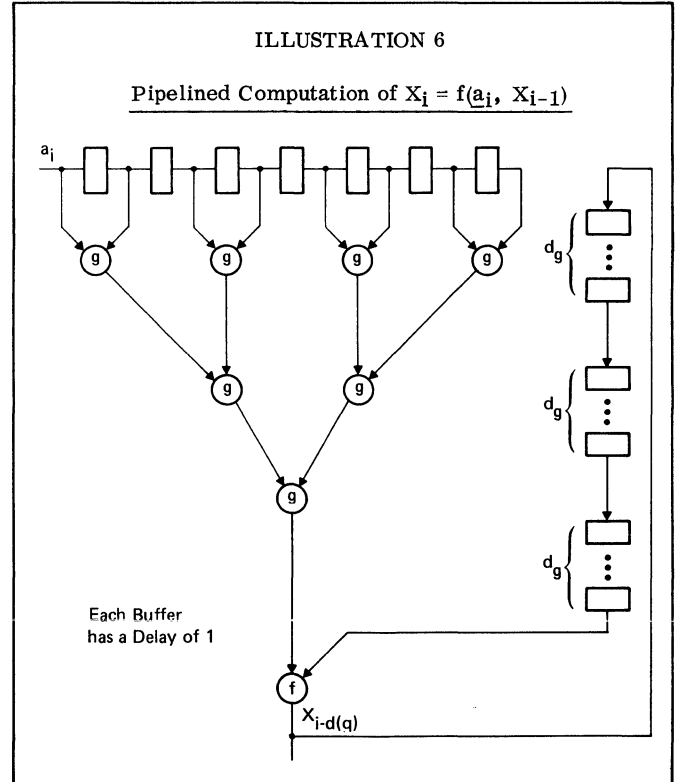
The output of the final  $g$  module drives the module that computes  $f$ , as pictured in Illustration 6. Again buffers are used to synchronize the arrival of data at each module. The

total delay through this pipeline is thus

$$d(q) = d_f + d_g \lceil \log_2 q \rceil \quad (12)$$

Again a maximum rate pipeline requires that

$$d(q) = d_f + d_g \lceil \log_2 q \rceil \leq q \quad (13)$$



As with Illustration 4, many of the  $g$  modules in Illustration 6 are redundant in that the computations they perform are identical to computations performed several time units earlier by some other module. Consequently, they can be replaced by buffers that simply delay the output from the other module by the appropriate amount. If  $q$  is chosen to be the minimum integer power of two that satisfies equation 13, then this technique of substituting buffers for  $g$  modules reduces a network like Illustration 6, containing  $q-1$   $g$  modules, into one like Illustration 7, which uses only  $\lceil \log_2 q \rceil$  modules.

Table 1 summarizes, as a function of  $d_f$  and  $d_g$ , the minimum  $q$  ( $q_{min}$ ) that satisfies equation 13, the corresponding  $d(q)$ , the minimum  $q$  that is a power of two ( $2^{\lceil \log_2 q_{min} \rceil}$ ), and the number of  $g$  modules ( $\lceil \log_2 q \rceil$ ).

### IV. M<sup>th</sup>-ORDER RECURRENCES

An  $m^{\text{th}}$ -order recurrence,  $m > 1$ , has the form

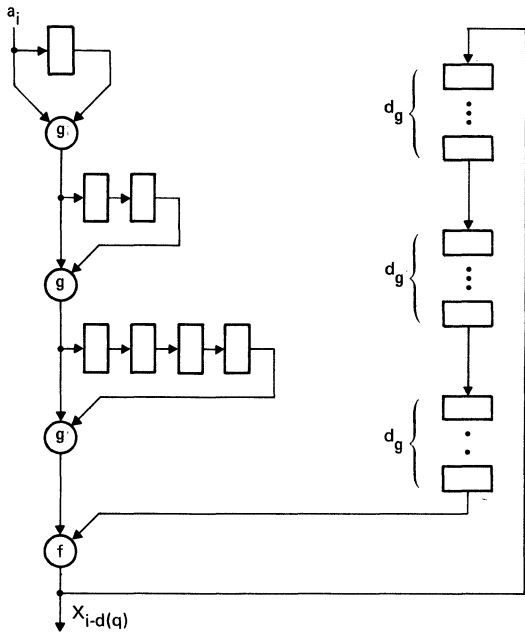
$$X_i = f(\underline{a}_i, X_{i-1}, \dots, X_{i-m}) \quad (14)$$

To speed up a pipeline computing this type of recurrence, we want to express  $X_i$  as

$$X_i = f(\underline{a}_i^*, X_{i-q}, \dots, X_{i-q-m+1}) \quad (15)$$

ILLUSTRATION 7

Minimal Complexity Pipeline for  $X_i = f(\underline{a}_i, X_{i-1})$



where the time to compute  $\underline{a}_i^*$  from the original  $\underline{a}_i$ 's grows less rapidly than, and eventually is smaller than  $d$ . In the  $m^{\text{th}}$ -order case, no simple associative or semi-associative companion function is possible; the number of arguments ( $\geq 2$ ) is too large. However, as was shown in earlier reports, many common recurrence functions have a related pair of functions that do allow the construction of networks with the desired characteristics(1,2). These are defined as follows:

DEFINITION 2

A recurrence function,  $f$ , is said to have a companion set  $(g, h)$  if there exists functions  $g$  and  $h$  such that for all parameter vectors  $\underline{a}_0, \dots, \underline{a}_m$  and all  $X$ 's,  $X_1 \dots X_m$ :

$$f(\underline{a}_0, f(\underline{a}_1, X_1, \dots, X_m), X_1, \dots, X_{m-1}) = f(g(\underline{a}_0, \underline{a}_1), X_1 \dots X_m) \tag{16}$$

$$f(\underline{a}_0, f(\underline{a}_1, X_1, \dots, X_m), \dots, f(\underline{a}_m, X_1, \dots, X_m)) = f(h(\underline{a}_0, \underline{a}_1, \dots, \underline{a}_m), X_1, \dots, X_m) \tag{17}$$

As an example, the  $m^{\text{th}}$ -order linear recurrence (equation 2) has the following companion set (where  $g(\underline{a}, \underline{b})$  stands for

TABLE 1

Complexity of Pipelines for First-Order Recurrences

$d_f \backslash d_g$	2	3	4	5	6	7	8	9	10
1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1	1 0 1 1
2	8 3 8 8	14 4 16 14	22 5 32 22	27 5 32 27	32 5 32 32	44 6 64 44	50 6 64 50	56 6 64 56	62 6 64 62
3	11 4 16 11	15 4 16 15	23 5 32 23	28 5 32 28	39 6 64 39	45 6 64 45	51 6 64 51	57 6 64 57	63 6 64 63
4	12 4 16 12	16 4 16 16	24 5 32 24	29 5 32 29	40 6 64 40	46 6 64 46	52 6 64 52	58 6 64 58	64 6 64 64
5	13 4 16 13	20 5 32 20	25 5 32 25	30 5 32 30	41 6 64 41	47 6 64 47	53 6 64 53	59 6 64 59	75 7 128 75
6	14 4 16 14	21 5 32 21	26 5 32 26	31 5 32 31	42 6 64 42	48 6 64 48	54 6 64 54	60 6 64 60	76 7 128 76
7	15 4 16 15	22 5 32 22	27 5 32 27	32 5 32 32	43 6 64 43	49 6 64 49	55 6 64 55	61 6 64 61	77 7 128 77
8	16 4 16 16	23 5 32 23	28 5 32 28	38 6 64 38	44 6 64 44	50 6 64 50	56 6 64 56	62 6 64 62	78 7 128 78
9	19 5 32 19	24 5 32 24	29 5 32 29	39 6 64 39	45 6 64 45	51 6 64 51	57 6 64 57	63 6 64 63	79 7 128 79
10	20 5 32 20	25 5 32 25	30 5 32 30	40 6 64 40	46 6 64 46	52 6 64 52	58 6 64 58	64 6 64 64	80 7 128 80

Each Entry:  $q \begin{bmatrix} \log_2 q \\ 2 \log_2 q \end{bmatrix} d(q)$

$j^{\text{th}}$  component of the parameter vector  $g(\underline{a}, \underline{b})$ :

$$g(\underline{a}, \underline{b})(j) = \begin{cases} \underline{a}(1)\underline{b}(j) + \underline{a}(j+1) & 1 \leq j \leq m-1 \\ \underline{a}(1)\underline{b}(m) & j = m \\ \underline{a}(m+1) + \underline{a}(1)\underline{b}(m+1) & j = m+1 \end{cases} \quad (18)$$

$$h(\underline{a}_0, \dots, \underline{a}_m)(j) = \begin{cases} \sum_{r=1}^m \underline{a}_0(r) \underline{a}_r(j) & 1 \leq j \leq m \\ \sum_{r=1}^m \underline{a}_0(r) \underline{a}_r(m+1) + \underline{a}_0(m+1) & j = m+1 \end{cases} \quad (19)$$

The utility of companion functions comes from the following theorem (proved in reference 1):

**THEOREM 2**

For any  $K \geq 0$ ,  $X_i$  can be expressed in terms of  $X_{i-q(k)}$  as follows

$$X_i = f(\underline{a}_i^{(k)}, X_{i-q(k)}, \dots, X_{i-q(k)-m+1}) \quad (20)$$

where

$$q(k) = m2^k + 1 - m \quad (21)$$

and  $\underline{a}_i^{(k)}$  is computed from the following recurrence:

$$\underline{a}_i^{(0)} = \underline{a}_i \quad (22)$$

$$\underline{a}_i^{(k+1)} = h(\underline{a}_i^{(k)}, A(i-q(k), m-1), A(i-q(k)-1, m-2), \dots, A(i-q(k)-m+1, 0)) \quad (23)$$

$$A(i, j) = g(g(\dots g(\underline{a}_i^{(k)}, \underline{a}_{i-q(k)}^{(k)}), \underline{a}_{i-q(k)-1}^{(k)}), \dots, \underline{a}_{i-q(k)-j+1}^{(k)}) \quad (24)$$

Illustration 8 diagrams a typical network for computing  $\underline{a}_i^{(k+1)}$  from  $\underline{a}_i^{(k)}$ . Since the function  $g$  is not usually associative, or even semi-associative, no rearrangement or reduction in the number of  $g$  modules is generally possible.

The total delay in Illustration 8 is thus:

$$(M-1) d_g + d_h \text{ time units} \quad (25)$$

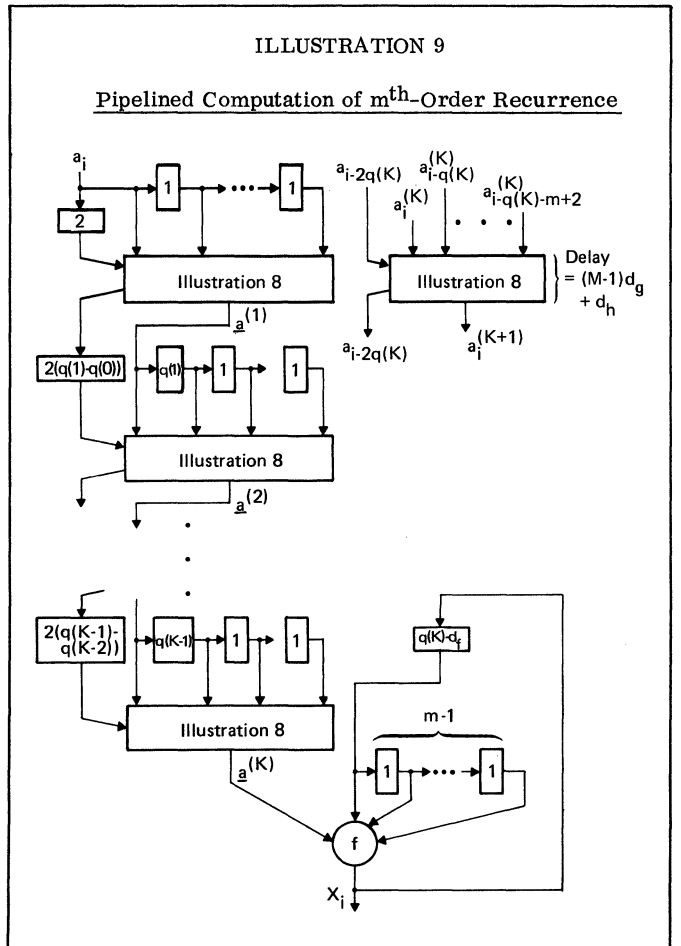
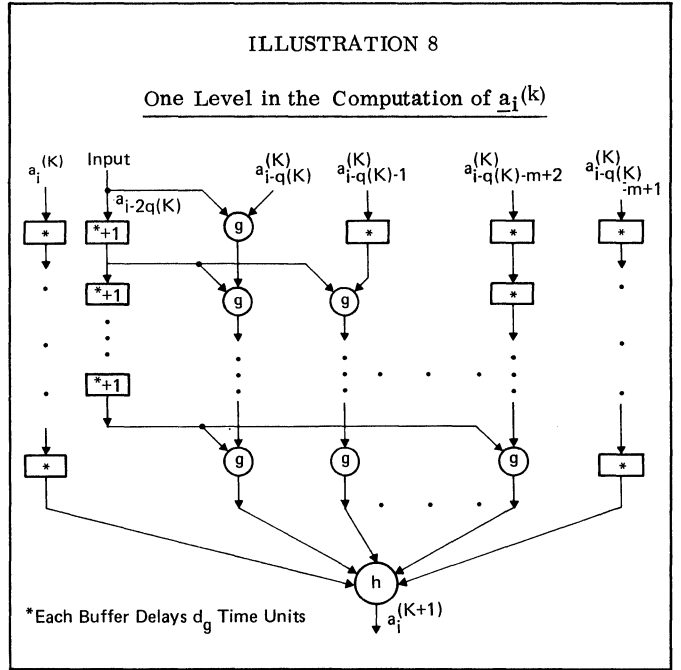
To build a pipeline that computes  $\underline{a}_i^{(k)}$  directly from the  $\underline{a}_j$ 's, the network of Illustration 8 must be cascaded into  $K$  levels. As with the earlier pipelines, however, only one copy of Illustration 8 is needed at each level. Additional buffers are used to save redundant computations and synchronize the arrival of the proper inputs. Illustration 9 diagrams such a pipeline.

For a  $K$ -level pipeline, like Illustration 9, the total delay through the pipeline is simply  $K$  times the delay of a single network (equation 5) plus the delay to compute  $f$ :

$$d(K) = K((m-1) d_g + d_h) + d_f \quad (26)$$

Again, for a maximal rate pipeline, this delay must be less than  $q(k)$ , equation 21; i. e., a  $K$  must be found such that

$$K((m-1) d_g + d_h) + d_f \leq m2^K - m + 1 \quad (27)$$



Once this minimal value of  $K$  has been determined, the complexity of the required pipeline can be computed directly from Illustrations 8 and 9, as shown in Table 2.

TABLE 2	
Complexity of Pipelines for $m^{\text{th}}$ -Order Recurrences	
Type of Module	Number Required*
f	1
h	$K$
g	$K \frac{m(m-1)}{2}$

\* $K$  is smallest positive integer that satisfies equation 27.

## V. CONCLUSIONS

This paper has discussed methods of speeding up pipelined computation of recurrence problems where feedback is present; that is, where the computation of one element of the desired sequence cannot be started before some earlier element has been fully computed. The methods discussed basically involve rewriting the recurrence so that  $X_i$  depends on

$$X_{i-q(k)}, \dots, X_{i-q(k) - m+1} \quad (28)$$

and some computable parameter vector  $\underline{a}_i^{(k)}$ . For many recurrence problems, the time to compute  $\underline{a}_i^{(k)}$  grows much less rapidly with  $k$  than does  $q(k)$ . In such circumstances, for large enough  $k$ , the total time to compute  $X_i$  from  $X_{i-q(k)}, \dots$  is less than  $q(k)$ , allowing the output of the resulting pipeline to be fed directly back into the input and yet still maintain a fully utilized pipeline that outputs a new  $X_i$  during each time unit. This pipeline is then running at the maximum possible rate.

One question that has not been discussed in detail in this paper is the problem of initializing the pipeline. The most direct techniques would be simply to precompute enough  $X_i$ 's and  $\underline{a}^{(j)}$ 's to fully initialize all stages in the pipeline (perhaps using parts of the same pipeline at less than maxi-

mal rate). Once this is done, the pipeline can be allowed to run normally. This is a time-consuming process which, if the pipeline is long enough, may negate many of the benefits of the maximal rate pipeline once it is started. For some specific problems, however, this process may be avoidable by introducing special values for  $\underline{a}_i$ 's and  $X_i$ 's. For example, in Illustration 5, if the input to B6 is held to 1 for the first 6 time units, and B1 initially loaded with 1, the pipeline will output  $X_1$  at time 6, and run normally after that.

The applicability of these speedup techniques depends in large measure on the particular problem being solved, the length of the desired solution sequence, and the stage delays in the basic f, g, and h computing modules. For problems where the modules have large stage delays, for example, the potential maximum speedup is significant, but the value of  $k$  required to attain that speedup may result in a very long and complex pipeline, where the time to initialize the pipeline becomes a significant fraction of the total computation time. In such cases, some kind of iterative tradeoff between changing module stage delays, accepting less than maximal output rates, and initializing the pipeline may be necessary.

## REFERENCES

1. Kogge, P. M. "The Parallel Solution of Recurrence Problems," PhD Thesis. Stanford University, December 1972. (To be published in IBM Journal of Research and Development, March 1974.)
2. Kogge, P. M. "The Parallel Solution of Recurrence Problems," 7th Annual Princeton Conference on Information and System Sciences. Princeton University, March 1973. (This is a partial summary of Reference 1.)
3. Kogge, P. M. and Stone, H. S. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," IEEE Transactions on Computers, August 1973.
4. Kuck, D. "ILLIAC IV Software and Applications Programming," IEEE Transactions on Computers, Vol. C-17, No. 8, August 1968, pp. 758-770.



# COMMENTS ON CAPABILITIES, LIMITATIONS AND "CORRECTNESS" OF PETRI NETS\*

Tilak Agerwala  
Mike Flynn

Electrical Engineering Department  
The Johns Hopkins University  
Baltimore, Maryland

## ABSTRACT

In this paper we examine the capabilities and limitations of Petri nets and investigate techniques for proving their correctness. We define different classes of nets where each is basically a Petri net with slight modifications and study the relationship between the various classes. One particular class appears to be quite powerful, with respect to its capability for representing coordinations. In the second part of the paper we establish the feasibility of using the methods of computational induction and inductive assertions to prove restricted statements about Petri nets.

## I. INTRODUCTION

Petri nets are being widely used in the design, specification and evaluation of computer systems [1,7], and in the modeling of production [3] and legal [6] systems. They also appear to be a neat, clear and convenient way to express process coordination. Naturally, the question about capabilities and limitations of these nets arises. It has been shown [4] that there are problems where the desired coordination cannot be expressed using Petri nets. In the first part of this report we introduce different classes of nets. Each class is basically a Petri net with slight modifications. We then examine the relationship between the various classes in the hope that this will give us some insight into the capabilities and limitations of Petri nets.

In the second part we are concerned with proving assertions about Petri nets. Given a coordination problem and a Petri net it should be possible to convince oneself that the Petri net does in fact represent the desired coordination correctly. Techniques for proving any given Petri net correct, will help in proving the correctness of general parallel systems since it may be possible to translate the system mechanically into a Petri net where it is easier to see what is going on.

## II. CAPABILITIES AND LIMITATIONS

We assume that the reader is familiar with Petri nets and concepts such as liveness, safety, etc. However, for the sake of avoiding ambiguity we will define a Petri net and give the simulation rules explicitly.

A Petri net  $N$  is a directed graph defined as a quadruplet  $(T, P, A, M^0)$  where,

- $T = \{t_1, \dots, t_n\}$  is a finite set of transitions
- $P = \{p_1, \dots, p_m\}$  is a finite set of places  
( $T, P$  form the nodes of the graph)
- $A = \{a_1, \dots, a_k\}$  is a finite set of directed arcs

of the form  $(x, y)$  which either connect a transition to a place or a place to a transition. Each place may have one or more markers in it or it may be empty. A place is full if it has at least one marker.

$$M^0 = \{(p, n) \mid p \in P \text{ and } n \in \{0, 1, 2, \dots\}\}$$

(a function from  $P$  to  $\{0, 1, 2, \dots\}$ ) is the initial marking.

### Simulation Rules

Given a certain marking  $M$  of a net, if all the input places to a transition are full the transition is said to be enabled in  $M$ . An enabled transition may at some stage decide to fire. At this stage it reserves a marker in each input place and starts firing. At the completion of firing it removes the reserved markers and places a marker in each output place, giving a new marking  $M'$ . We say that the firing of  $t_i$  in  $M$  results in  $M'$ . As soon as a marker is reserved it becomes invisible to all other transitions.

$$\bar{t} = t_{b_1}, t_{b_2}, \dots, t_{b_n} \in T^*$$

is said to be a simulation sequence of a net  $N = (T, P, A, M^0)$  if there exists a sequence of markings  $M^0, M^1, \dots, M^n$  such that  $t_{b_i}$  is enabled in  $M^{i-1}$  and firing of  $t_{b_i}$  in  $M^{i-1}$  results in  $M^i$ , for all  $i \in \{1, 2, \dots, n\}$ . The set of all simulation sequences of  $N$  is called the simulation set of  $N$  or  $\text{SIMSET}_N$ . Let  $T' \subseteq T$ . Then for each simulation sequence  $\bar{t} = t_{b_1}, \dots, t_{b_n}$  of  $N$  we define a reduced simulation sequence  $\bar{t}' = t_{c_1}, t_{c_2}, \dots, t_{c_p}$  with respect to  $T'$ , where  $\bar{t}'$  is the sequence that results when all  $t_{b_i} \in T - T'$  are excluded from  $\bar{t}$ .  $\text{SIMSET} \mid T'$  is the set of all reduced simulation sequences of  $N$  with respect to  $T'$ . Two Petri nets  $N_1 = (T_1, P_1, A_1, M_1^0)$  and  $N_2 = (T_2, P_2, A_2, M_2^0)$  are said to be strongly equivalent with respect to  $T$  if  $T \subseteq T_1, T \subseteq T_2$  and  $\text{SIMSET}_{N_1} \mid T = \text{SIMSET}_{N_2} \mid T$ . In this case we write  $N_1 \stackrel{T}{=} N_2$ .

So far, we assumed that the transitions of Petri nets had distinct labels. We now define an interpretation  $I [T', E]$  of a Petri net  $N = (T, P, A, M^0)$  as follows:

$$T' = \{t_{a_1}, \dots, t_{a_m}\} \subset T \text{ is a set of transitions,}$$

$$E = \{E_1, \dots, E_k\}, k \leq m$$

is a set of event or process names and  $I: T' \rightarrow E$ , i.e.  $I$  is a function from  $T'$  onto  $E$ . Thus, the same event or process name may be attached to different transitions and the same net may represent different coordinations depending on the interpretation given to it. Given a net  $N = (T, P, A, M^0)$  and an interpretation  $I [T', E]$ , for

each reduced simulation sequence  $t_{b_1}, \dots, t_{b_m}$  with respect to  $T'$  we get an interpreted simulation sequence  $E_{c_1}, E_{c_2}, \dots, E_{c_m}$  with respect to  $I$  where  $I(t_{b_i}) = E_{c_i}$  for  $1 \leq i \leq m$ . The set of all interpreted sequences of  $N$  with respect to  $I$  is called  $I[\text{SIMSET}_N]$ . A net  $N_1$  with an interpretation  $I_1[T', E]$  is weakly equivalent to a net  $N_2$  with interpretation  $I_2[T'', E]$  if  $I_1[\text{SIMSET}_{N_1}] = I_2[\text{SIMSET}_{N_2}]$  and in this case we

$$N_1 \stackrel{I_1, I_2}{\equiv} N_2$$

In what follows we will define different classes of nets where each kind is basically a Petri net with slight modifications.  $\text{SIMSET}$ ,  $\text{SIMSET} \mid T$  and  $I[\text{SIMSET}]$  can be appropriately defined for each class. If  $\text{TN}$  and  $\text{TN}_x$  refer to two different classes of nets, then  $\text{PN}$  and  $\text{PN}_x$  refer to all the coordinations representable by  $\text{TN}$  and  $\text{TN}_x$  respectively. We say that  $\text{PN} \subseteq \text{PN}_x$  if for every  $N \in \text{TN}$  and interpretation  $I[T', E]$  there exists an  $N_x \in \text{TN}_x$  and interpretation  $I_x[T'', E]$  such that

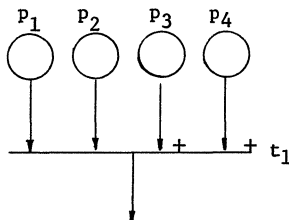
$$N \stackrel{I, I_x}{\equiv} N_x.$$

Thus  $\text{PN} \subseteq \text{PN}_x$  if  $\text{PN} \subseteq \text{PN}_x$  and there exists a net  $N_x \in \text{TN}_x$  and an interpretation  $I_x[T'', E]$  such that there is no net  $N \in \text{TN}$  and interpretation  $I[T', E]$  with

$$N \stackrel{I, I_x}{\equiv} N_x.$$

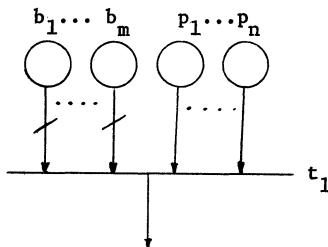
### Classes of Nets

- Let the class of ordinary Petri nets be  $\text{TN}$ .
- The transitions in ordinary Petri nets are enabled only when all the input places are full and we can consider these transitions to have an AND-input logic. If in addition, we allow transitions with OR input logic, we call the class of nets  $\text{TN}_{\text{log}}$




(letting  $P_i$  denote the number of markers in  $p_i$ )  $t_1$  is enabled if and only if  $[(P_1 > 0) \wedge (P_2 > 0)] \vee [(P_3 > 0) \vee (P_4 > 0)]$ . Thus  $t_1$  is enabled even if all the input places do not have markers and when it starts firing it reserves a marker in each input place that has at least one.

- In addition to the ordinary transitions in the nets belonging to  $\text{TN}$  we allow a transition to have input places and arcs of a special kind. The transitions allowed are of the form:



$t_1$  is enabled if and only if  $(B_1 = 0) \wedge (B_2 = 0) \wedge \dots \wedge (B_n = 0) \wedge (P_1 > 0) \wedge (P_2 > 0) \wedge \dots \wedge (P_n > 0)$ . When  $t_1$  starts firing a marker is reserved in each of  $P_1, P_2, P_3, \dots, P_n$ . Let the class of nets be called  $\text{TN}_{\text{com}}$ .

- In addition to the ordinary places in the nets belonging to  $\text{TN}$  we introduce a special place , (say  $p_1$ ). A transition will place a stone in  $p_1$  if and only if  $P_1 = 0$ . Let the class of nets be  $\text{TN}_{\text{out}}$ .

### Results

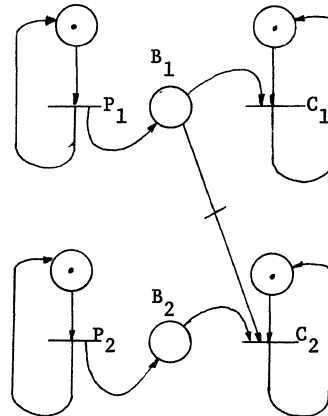
- Since in each case we provided the nets with additional capabilities over the nets belonging to  $\text{TN}$ , obviously:

$$\begin{aligned} \text{PN} &\subseteq \text{PN}_{\text{log}} \\ \text{PN} &\subseteq \text{PN}_{\text{out}} \\ \text{PN} &\subseteq \text{PN}_{\text{com}} \end{aligned}$$

- $\text{PN} \subseteq \text{PN}_{\text{com}}$

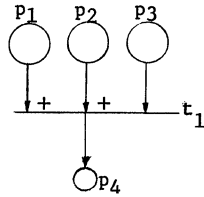
### Proof

Kosaraju [4] describes a coordination problem and proves that it falls outside  $\text{PN}$ . The problem is as follows: There are four cyclic processes,  $P_1, P_2, C_1$  and  $C_2$  and two buffers  $B_1$  and  $B_2$ .  $P_1$  and  $P_2$  are producers which place one item each on top of  $B_1$  and  $B_2$  respectively in every cycle.  $C_1$  and  $C_2$  consume one item each from the bottom of  $B_1$  and  $B_2$  respectively. However,  $C_1$  has higher priority than  $C_2$  so that  $C_2$  can consume only if  $B_1$  is empty. To prove that  $\text{PN} \subset \text{PN}_{\text{com}}$  we will give an interpreted net belonging to  $\text{TN}_{\text{com}}$  which represents the desired coordination. The net is:

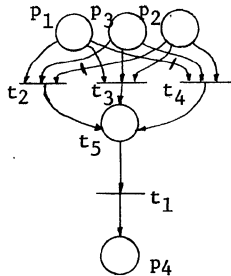


$$3a. \text{PN}_{\text{log}} \subseteq \text{PN}_{\text{com}}$$

For every net  $N_{\text{log}} = (T, P, A, M^0)$  there exists a net  $N_{\text{com}} = (T', P', A', M_1^0) \in \text{TN}_{\text{com}}$  such that  $T \subset T'$  and  $N_{\text{log}} \stackrel{T}{\equiv} N_{\text{com}}$ . The result 3a follows from this. We will not go into the details of a proof but will illustrate the idea by means of an example. Let the net  $N$  below be part of a larger net  $N_{\text{log}} = (T, P, A, M^0)$  belonging to  $\text{TN}_{\text{log}}$ .



N can be replaced by:



Let the resulting net be  $N'$ . Then obviously  $N' \stackrel{T}{=} N_{log}$ . By applying a similar procedure to each transition with OR input logic we end up with a net  $N_{com}$  which is strongly equivalent to  $N_{log}$  with respect to T.

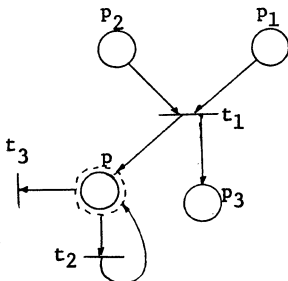
3b. Kosaraju's problem 1 and proof [4] can be used to prove that  $PN_{log} \subset PN_{com}$ .

4a.  $PN_{out} = PN$

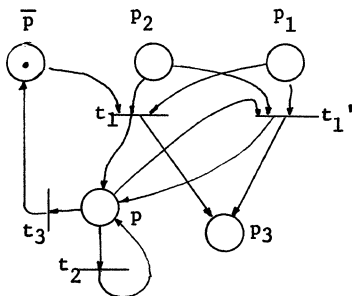
For every net  $N_{out} = (T_1, P_1, A_1, M_1^0) \in TN_{out}$  and interpretation  $I_1 [T, E]$ , there exists a net  $N = (T_2, P_2, A_2, M_2^0) \in TN$  and interpretation  $I_2 [T', E]$ , such that

$$N_{out} \stackrel{I_1, I_2}{=} N.$$

Again, we will not go into details of a proof but will illustrate with an example. Let the net N below be part of a larger net  $N_{out}$ .



Then N can be replaced by:



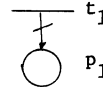
resulting in the net  $N'$ . Here we have introduced a place  $\bar{p}$  which is a complement of p in the sense that  $\bar{p}$  has a marker if and only if p does not. The reader can convince himself that under the interpretation  $I' [T \cup \{t_1'\}, E]$ , where  $I'(t) = I_1(t)$  for  $t \in T$  and

$I_2(t_1') = I_1(t)$ ,  $N_{out} \stackrel{I_1, I'}{=} N'$ . Continuing this process until all places of the form  $\bar{p}$  are eliminated, we end up with a net  $N \in TN$  and an interpretation  $I_2 [T', E]$  such that

$$N_{out} \stackrel{I_1, I_2}{=} N.$$

This shows that  $PN_{out} \subseteq PN$  and from result 1 we conclude that  $PN_{out} = PN$ . Since  $PN \subset PN_{com}$  we also conclude that  $PN_{out} \subset PN_{com}$ .

Comments: We feel that  $PN \subset PN_{log}$ . We are also examining other classes of nets. For example, in addition to the ordinary arcs between transitions and places we allow the following:

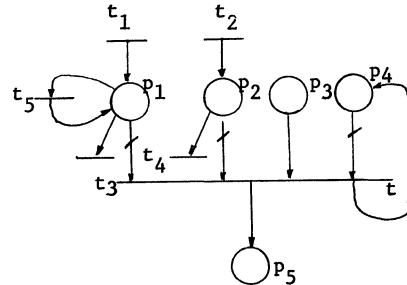


$t_1$  will place a marker in  $p_1$  if and only if  $P_1 > 0$ . Another class of nets is those where we allow a transition to nondeterministically place a marker in one or more of its output places. The results obtained so far indicate that  $TN_{com}$  is a very powerful class of nets.

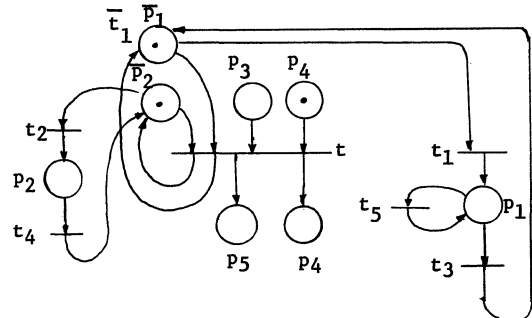
### Safe nets

If one considers only safe nets (where each place can contain at most one marker at any stage), then it can be shown that for every  $N_{com} = (T, P, A, M^0) \in TN_{com}$  that is safe, there exists a safe net  $N \in TN$  such that

$N_{com} \stackrel{T}{=} N$ . Again, we will only demonstrate the technique of obtaining N with the help of an example. Let the net  $N_1$  below be part of a safe member  $N_{com}$  of  $TN_{com}$ .



Replace  $N_1$  by the net below to get a net  $N'$



The fact that  $N_{com}$  is safe permits us to introduce places  $\bar{p}_1, \bar{p}_2, \bar{p}_4$  which are complements of the places  $p_1, p_2$  and  $p_4$  respectively. I.e.  $\bar{p}_1$  has a marker if and only if  $p_1$  does not. Every transition that causes a marker to be put in  $p_1$  should cause a marker to be removed from  $\bar{p}_1$ . Every transition that causes a marker to be removed from  $p_1$  should cause a marker to be placed in  $\bar{p}_1$ . We now have

$$N' \stackrel{T}{=} N_{com}.$$

By continuing the process of replacement we end up with a net  $N_k \in TN$  such that  $N_k \stackrel{T}{=} N_{com}$ . If  $PN_x | safe$  denotes the set of coordinations representable by safe members of  $TN_x$ , then  $PN | safe = PN_{com} | safe$ . From results 3 and 4  $PN | safe = PN_{log} | safe = PN_{out} | safe = PN_{com} | safe$ .

Thus, even though  $TN_{com}$  is a powerful class of nets, in practice one would probably be more concerned with safe nets and here the modifications made to ordinary Petri nets do not increase the overall power.

### III. CORRECTNESS

When we say that a "Petri net N is correct", intuitively what is meant is that the Petri net does what the designer intended it to do. Given a particular problem, a Petri net is constructed which represents the desired coordination. First and foremost we are not at all concerned with whether the Petri net is the best one for the given problem. In fact, we will not even try to prove that the Petri net effectively represents the desired coordination. We shall, however, try to prove very restricted statements about a net which are provided by the designer. The kinds of statements we will attempt to prove are:

1. At any given time only one of the transitions from the set  $\{t_1, \dots, t_k\}$  may be firing.
2. Two given transitions will never conflict.
3. A given place is safe with respect to a particular marking or a given marking is safe.
4. A given place can contain at most N markers
5. A given transition is live.
6. A given marking is reachable from another.
7. A given transition has fired at most x times.
8. In general it may be very difficult to show that a "net is deadlock free". Again, the designer will have to provide statements, for example, "Every transition in cycle C is live at every stage", from which he can reasonably conclude that the net will not hang up.

In the following we present two methods to prove the correctness of Petri nets: Computational induction and inductive assertions.

#### Computational Induction

Here we develop certain relations that remain invariant during the simulation of a net. By using these relations suitably we will be able to prove certain properties about the net. According to our simulation rules, when a transition starts firing it reserves a marker in each input place. Reserved markers are invisible to all other transitions. However, in the invariant relations, all reserved markers are also counted and assumed to be in their current places. The relations follow trivially from the simulation rules. Let,

- $M_i$ : Number of stores in  $p_i$  initially
- $P_i$ : Number of stores in  $p_i$  at any instant
- $T_i$ : Number of times  $t_i$  has fired till any instant.

**Relation 1:** Let  $I_i = \{\text{set of transitions with } p_i \text{ as output place}\}$ ,  $O_i = \{\text{set of transitions with } p_i \text{ as input place}\}$ , then

$$P_i = \sum_{t_k \in I_i} T_k - \sum_{t_j \in O_i} T_j + M_i \geq 0$$

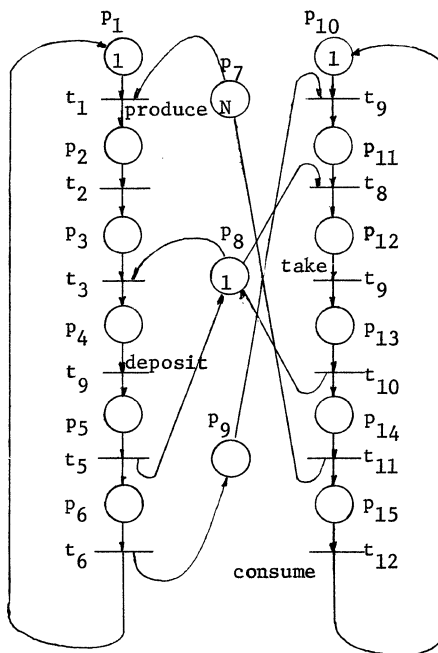
**Relation 2:** Let  $t_{a_1}, p_{b_1}, t_{a_2}, \dots, p_{b_{k-1}}, t_{a_k}$  be a path in the net such that  $t_{a_i}, p_{b_i} \quad 1 \leq i \leq k$  are distinct. If in addition  $I_{b_i} = \{t_{a_i}\}$ ,  $1 \leq i \leq k-1$  then

we have a simple path and  $T_{a_k} \leq T_{a_1} + \sum_{i=1}^{k-1} M_{b_i}$ .

**Relation 3:** If  $S_1$  is a simple path from  $t_i$  to  $t_j$  and  $S_2$  is a simple path from  $t_j$  to  $t_i$  then  $S_1 S_2$  forms a simple cycle. If in addition every place on a simple cycle has only one input and one output arc then we have a pure cycle. Let S be a pure cycle then:

$$\sum_{p_i \text{ in } S} P_i = \sum_{p_i \text{ in } S} M_i = N_s \quad (\text{say}),$$

We have used these relations to prove simple assertions about nets, and will illustrate the method by means of an example. Consider the producer consumer problem with bounded buffer. The producer places items in a buffer. (length N) and the consumer consumes them. The problem is to coordinate these two essentially independent processes so that the consumer does not try to take an item from the buffer when it is empty and the producer does not place an item where the buffer is full. The Petri net that represents the described coordination is given below: (the numbers in the places denote the initial number of markers)



We are interested in proving the following properties for this net:

1.  $t_4$  and  $t_9$  cannot be firing at the same time, i.e. the producer and consumer do not try to access the buffer at the same time.
2.  $0 \leq T_4 - T_9 \leq N$ . I.e. there is no buffer overflow or underflow.
3. the net is deadlock free.

#### Proof 1:

$$T_8 + T_3 \leq 1 + T_{10} + T_5 \quad (1) \text{ By } R_1$$

$$P_4 = T_3 - T_4 \quad (2) \text{ By } R_1$$

$$T_5 \leq T_4 \quad (3) \text{ By } R_2$$

$$P_4 \leq T_3 - T_5 \quad (4) \text{ from (2) + (3)}$$

Similarly,  $P_{12} \leq T_8 - T_{10}$  (5)  
 Therefore,  $P_4 + P_{12} \leq 1$  (6) from (4), (5), (1)  
 From 6 and the simulation rules we conclude directly  
 that  $T_4$  and  $T_9$  cannot be firing at the same time.

Proof 2.

$$T_4 \leq T_9 + N \quad (1) \text{ By } R_2$$

Therefore,  $T_4 - T_9 \leq N$   
 i.e., the number of deposits - the number of removals  
 $\leq N$ . Therefore, there can be no buffer overflow.

$$T_9 \leq T_4 \quad (2) \text{ By } R_2$$

Therefore,  $T_4 - T_9 \geq 0$   
 i.e., number of deposits - number of removals  $\geq 0$ .  
 Therefore there can be no buffer underflow.

Proof 3.

For this particular problem it is easy to see that dead-  
 lock can occur only if  $P_7 = P_9 = 0$  and there is no way  
 to change this situation. (pure cycles can be repre-  
 sented by the subscripts of the places only since there  
 is no ambiguity)

- $S_1 = 1, 2, 3, 4, 5, 6, 7, 1$  is a pure cycle
- $S_2 = 10, 11, 12, 13, 14, 15, 10$  is a pure cycle
- $S_3 = 3, 4, 5, 6, 9, 11, 12, 13, 14, 7, 3$  is a pure cycle
- $NS_1 = 1$  (1)
- $NS_2 = 1$  (2)
- $NS_3 = N$  (3)
- $a = P_3 + P_4 + P_5 + P_6 \leq 1$  (4) from (1)
- $b = P_{11} + P_{12} + P_{13} + P_{14} \leq 1$  (5) from (2)

Therefore,  $a + b \leq 2$  from (4), (5)

But if  $N > 2$  and  $P_7 = P_9 = 0$  then

$$a + b = N > 2 \quad \text{from (3)}$$

Therefore, we get a contradiction. Thus, for  $N > 2$ , at  
 no stage can both  $P_7$  and  $P_9$  be zero. Therefore, there  
 can be no deadlock for  $N > 2$ . For  $N = 1$  and  $N = 2$  separ-  
 ate arguments can be given to prove that the net is  
 deadlock free.

Inductive Assertions

This method was introduced by Floyd [2] to prove the  
 correctness of sequential programs and the same tech-  
 nique was used by Lauer [5] for proving parallel pro-  
 grams correct. We have taken the basic ideas from [5]  
 and modified them to be applicable in the framework of  
 Petri nets. Here again, our aim is to prove that a  
 Petri net is correct with respect to a particular given  
 assertion A. The procedure is as follows: with each  
 transition in the net we associate an assertion. Our  
 aim is to prove that every time a transition is enabled,  
 the corresponding assertion is true irrespective of the  
 particular simulation which caused this transition to  
 be enabled and irrespective of the state of the rest of  
 the net. Once this has been established, the truth of  
 A has to be deduced from the assertions at the transi-  
 tions.

Let  $N = (T, P, A, M^0)$  be a Petri net. An assertion  $a_i$   
 asserted with a transition  $t_i \in T$  is a predicate on the  
 values of  $P_k$  and  $T_k$  where  $P_k \in P$  and  $T_k \in T$ . The Petri  
 net is correct with respect to the assertion  $a_i$  if and  
 only if for each simulation of the net that enables  $t_i$ ,  
 $a_i$  is true when  $t_i$  is enabled. The net N is correct  
 with respect to a set of assertions if and only if it is  
 correct with respect to each assertion in the set. Let  
 $I_1 =$  set of input places of  $t_1$  and  $O_1 =$  the set of out-

put places. Then we have the following:

Induction Theorem

To prove that a Petri net  $N = (T, P, A, M^0)$  is correct  
 with respect to a set of assertions  $\{a_i | t_i \in T\}$  it is  
sufficient to prove the following:

(1)  $a_i$  is true for all  $t_i$  that are enabled in  $M^0$ .

(2) For each  $t_i \in T$ , let  $P_i = \{p | p \in I_i \wedge (p, 0) \in M^0\}$   
 i.e., the set of all initially unmarked input places of  
 $t_i$ . Let  $P_i = \{q_1, q_2, \dots, q_n\}$ . Let  $T_j = \{t_k | q_j \in O_k\}$ ,  
 $1 \leq j \leq n$ , i.e., the set of all transitions of which  $q_j$   
 an output place. Let  $B_i = \{(b_1, b_2, \dots, b_n) | t_{b_j} \in T_j\}$   
 Each n-tuple in  $B_i$  gives the set of transitions which  
 when fired cause markers to be placed in the initially  
 unmarked input places of  $t_i$ . Let  $\text{Fire}(b_1, \dots, b_n)$   
 denote the fact that the transitions  $t_{b_1}, \dots, t_{b_n}$  fire.  
 Then for each  $t_i \in T$ ,

$$a_{b_1} \wedge a_{b_2} \wedge \dots \wedge a_{b_n} \wedge \text{Fire}(b_1, \dots, b_n) \Rightarrow a_i$$

for all  $(b_1, b_2, \dots, b_n) \in B_i$  .... (1)

Proof: Obvious

Each equation of the form (1) is called a verification  
 condition. It should be clear to the reader that the  
 verification conditions are really very strong. Thus  
 the conditions are not necessary but only sufficient.

To prove a net correct, one may often have to construct  
 an augmented net. Let  $N_1 = (T_1, P_1, A_1, M_1^0)$  be a Petri  
 Net. Then  $N_2 = (T_2, P_2, A_2, M_2^0)$  is an augmentation of  
 $N_1$  if and only if  $T_1 \subseteq T_2$ ,  $P \subseteq P_2$ ,  $A_1 \subseteq A_2$ ,  $M_1^0 \subseteq M_2^0$   
 and  $N_1 \equiv N_2$ .

One can show that, if  $N_2$  is an augmentation of  $N_1$  then  
 $N_2$  is correct with respect to  $a_i$  where  $t_i \in T_1$  if and  
 only if  $N_1$  is correct with respect to  $a_i$ . Here  $a_i'$  is  
 the same as  $a_i$  with all references to  $t \in T_2 - T_1$  and  
 $p \in P_2 - P_1$  deleted.

Thus, to prove that a Petri net  $N = (T, P, A, M^0)$  is cor-  
 rect with respect to an assertion A one goes through  
 the following steps:

1. Formulate the assertion  $a_i$  for each transition  $t_i$ .
2. Prove that all assertions associated with transi-  
 tions that are initially enabled are true.
3. Prove that all the pertinent verification conditions  
 hold and conclude that N is correct with respect to  
 $\{a_i | t_i \in T\}$ . (Instead of (2) and (3) one may construct  
 an augmented net N' of N, associate appropriate assertions  
 with the transitions of N', carry out (2) and (3)  
 for N' and conclude that N is correct with respect to  
 $\{a_i | t_i \in T\}$ )
4. Deduce that the net operates correctly with respect  
 to the main overall assertion, A.

We have used this method to prove the correctness of a  
 Petri net representation of the producer - consumer  
 problem with respect to an overall assertion. Since the  
 assertions and proof are essentially similar to those of  
 Lauer [5] we will not present the example here. In a  
 subsequent report we will present weaker verification  
 conditions, examine whether it is necessary to associate  
 assertions with each and every transition and develop  
 "local" conditions under which places, arcs and tran-  
 sitions can be added to a net N resulting in an augu-  
 mented net N'.

#### IV. CONCLUSIONS

We hope that the discussion in Part II sheds some light on the capabilities and limitations of Petri nets.  $TN_{com}$  seems to be a powerful class of nets. It is possible that these nets do provide a correct, formal counterpart to the vague notion of a "coordination problem". We will examine this aspect in another report. Also, Petri nets seem to be sufficiently powerful if one is concerned only with safe nets. This may very well be the case in practice.

In part III we have established the feasibility of using the methods of computational induction and inductive assertions to prove restricted kinds of statements about Petri nets. Ultimately, work in this direction will facilitate the process of convincing oneself that a general concurrent system is correctly coordinated.

#### V. REFERENCES

1. Dennis, Jack, B. "Modular, Asynchronous Control Structures for a High Performance Processor", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970, pp. 55-80.
2. Floyd, R. W. "Assigning Meanings to Programs", Proceedings of a Symposium in Applied Mathematics, Vol.19, Mathematical Aspects of Computer Science, American Mathematical Society, 1967, pp. 19-32.
3. Hack, Michel "Analysis of Production Schemata by Petri nets", M.S. Th., Dept. of Electrical Engineering, MIT, Cambridge, Mass., February 1972.
4. Kosaraju, S. Rao "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", Hopkins Computer Research Rpt. #25, Research Program in Computer Systems Architecture, J.H.U., Balto., Md., May 1973.
5. Lauer, Hugh Conrad "Correctness in Operating Systems", Ph.D. Thesis, Carnegie-Mellon University, September 1972. AFOSR -TR- 72 - 2361, Contract F 44620 - 70 - C - 0107.
6. Meldman, Jeffery and Holt, Anatol "Petri Nets and Legal Systems", Jurimetrics Journal 12, December 1971, pp. 65-75
7. Noe, Jerre D., "A Petri Net Model of the CDC 6400" Proceedings of the ACM/SIGOPS Workshop on Systems Performance Evaluation, April 1971, pp. 362-378.

#### ACKNOWLEDGEMENT

I am grateful to my advisor, Professor Michael J. Flynn, for his continued help, guidance, encouragement and support. I would also like to thank my colleague, Joe Davison, for his helpful comments.

# FLOWWARE—A FLOW CHARTING PROCEDURE TO DESCRIBE DIGITAL NETWORKS

Dr. Wayne E. Omohundro  
MTS, Bell Labs  
and  
Dr. James H. Tracey  
University of Missouri

## ABSTRACT

FLOWWARE is an interactive, graphical language to aid in the understanding and design of digital networks. The language is based upon the concept of flow charting. The user specifies the register layout of the network and the sequential operation in the form of a flow chart on a graphics terminal. The flow chart allows a user who is unfamiliar with the network to easily understand the function and operation of the network.

## I. INTRODUCTION

Many languages [1-21] exist which aid the user in the specification and design of digital networks but they do not aid the user who is unfamiliar with the network in his attempt to understand the function and operation of the network. Flow charting of a program has been recognized as an easy method to help in the understanding as well as the debugging of a program. Therefore, since digital networks in many ways resemble a program (especially on the register transfer level), flow charting should aid the user in the understanding of digital networks.

Graphical languages using a flow charting concept have been proposed by Rouse [17] and Bell, et al. [18] but, to this date, they have not been implemented on a computer. Also Digital Equipment Corporation has a modular computer, the PDP-16, whose functions can be specified by the purchaser through a special purpose language called CHARTWARE [20].

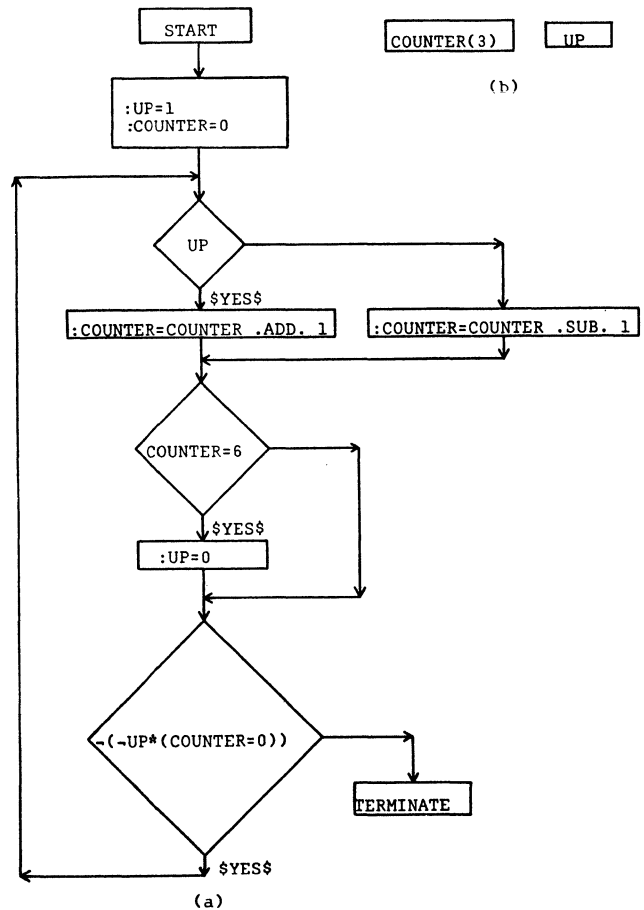
FLOWWARE [22-24] is an interactive, graphics language which allows the user to define a digital network in a manner similar to flow charting. The user can specify both the register layout as well as the sequential behavior of the network by using a flow chart and "drawing" the network on a graphics terminal. Consider a simple problem of specifying on a functional level a counter to count up from zero to six and then back down to zero. Figure 1 is an example of the flow chart necessary to describe this system. The elements are a three bit register COUNTER and a control signal UP. The blocks START and TERMINATE specify, respectively, the beginning and end points of the description. The reader should recognize that figure 1 represents exactly the information that the user would specify on the graphics terminal. Each rectangle, arrow, and diamond represents an element of the language FLOWWARE.

Figure 1 also shows some of the characteristics of the language. It is a graphical language and hence, gives the user a pictorial view of the network. The user specifies the register layout and can show the paths available for data transfer, the control signals regulating these transfers, and the functions which modify the data. Also, the sequential operation of the network

is shown in a graphical manner by a flow chart. The language is a means of specifying the functional behavior of a system without regard for the technology used for hardware implementation. FLOWWARE has been developed with the understanding that such problems as races, hazards, interconnection layouts, and fault analysis are not to be solved with this system. Its main purpose is to aid understanding but it can also be used in the initial phases of design when ideas are at the block diagram and functional level.

FIGURE 1

An Up/Down Counter (a) Control Flow Phase  
(b) Register Layout Phase



To assist the user, FLOWWARE is interactive and allows simulation of a design. The interactive nature permits the user to obtain his results immediately from a simulation run. Simulation helps a user understand a

network. He can change inputs and control signals to see what effect they have on the network, if he so desires, and resimulate it. In other words, the user interacts with a network to understand it or to verify its operational correctness.

FLOWWARE makes use of the IDDAP (Interactive Digital Design Assistance Package) system as written by Crall [15]. IDDAP is an interactive language which is a subset of Chu's CDL [10]. As such, it is oriented to text input rather than graphical input. Essentially, a preprocessor to handle the graphics information was added to IDDAP. There were several reasons for using IDDAP, one of which was that IDDAP is already an interactive system. Also IDDAP has a simulator to allow the description to be simulated. This was considered important because it makes it easier to verify that the system is working correctly, and also, as already mentioned, a simulator is useful in understanding the operation of a network. Finally, IDDAP handles translation of text input. In spite of the graphics nature of FLOWWARE, it is necessary to describe some operations by register transfer statements. Hence in order to concentrate on the graphics portion, rather than a simulator and text translator, a preprocessor was added to IDDAP. The major purpose of the preprocessor is the interconnect the graphical elements in the correct manner as specified by the user.

Often, when a person is describing a digital network informally, he draws a register layout to give an overall view of the system. Then he inserts the control signals and explains the sequential operation of the network. FLOWWARE formalizes this process. FLOWWARE relieves the user from drawing the elements. It forces the user to specify the register layout. It allows the user to use a graphical input in the form of a flow chart to specify sequential operations. The exact procedure and elements to perform these functions will now be explained.

## II. FLOWWARE ELEMENTS AND COMMANDS

This chapter presents the elements and commands of FLOWWARE. FLOWWARE has two description phases. Phase one is the register layout or information flow phase which serves to define the various components and show how they are interconnected. This phase is similar to the variable declaration statements of most languages but has the advantage of giving a pictorial view of the system. Phase two is the control flow phase. This phase makes use of the definitions in phase one to describe the data and control flow of the system by specifying a flow chart. The flow chart gives the sequence in which functions and decisions are activated along with the control signals regulating the events.

Table 1 presents the elements and commands as well as a brief description of their purpose. Each element is drawn by the computer on a graphics terminal at a position specified by the hand movement of a cursor using a joystick or mouse. Elements are defined by positioning the graphics cursor at the major defining point and typing the appropriate command. The major defining point is that graphics point which denotes the position at which the element is to be drawn by the computer. Some elements have a minor defining point because two points are necessary to define that element; for example, a line. Most of the elements have some text associated with them. The text defines the additional information needed for the element. In many cases this is the name by which the element is to be referenced, or some function to be performed. Editing features are also provided to allow the user to add, change, or delete elements or text.

TABLE 1  
Elements and Commands (a) Phase One

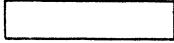
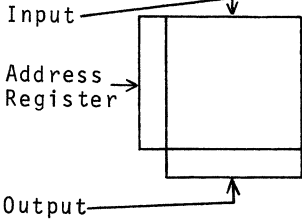
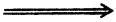


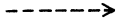

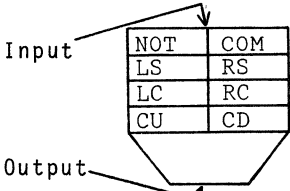
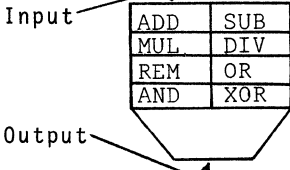



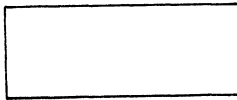
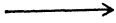

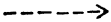
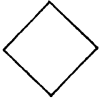
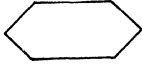
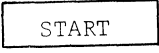
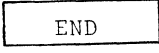

Meaning	Computer Response
Define <u>Register</u>	
Define <u>Memory</u>	
Define <u>Information flow connector</u>	
Define <u>Function</u>	
Define <u>Control signal</u>	
Define <u>control flow Line</u>	
Define <u>dEcodEr</u>	
Define <u>Unary operand function element</u>	
Define <u>Binary operand function element</u>	
Define <u>clock</u>	
Define <u>terminal</u>	
Define <u>Subregister</u>	



TABLE 1  
Elements and Command (b) Phase Two

Define <u>F</u> unction block	
Define <u>G</u> o to line	
Define <u>C</u> ontrol signal	
Define control flow <u>L</u> ine	
Define <u>D</u> ecision block	
Define <u>d</u> ecod <u>E</u> block	
Define start block	
Define end block	
Define terminate block	

A. PHASE 1 OR INFORMATION FLOW PHASE ELEMENTS

Phase 1 or information flow phase is used to define the components of the digital system to be simulated, to describe the register layout of the system, and to show the functions to be performed on the data. The basic elements of this phase are given in the following sections.

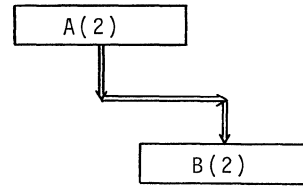
1. Register and Memory

The register and memory are common elements of a computer. As such the user can define a register and its length as well as a memory with its word length and the number of words. Table 1 shows the memory element. The input and output points specify those points by which the memory is accessed for writing and reading respectively. The memory address register, defined as a portion of the memory element, specifies the word of memory to be accessed.

2. Information Flow Connector

The information flow connector is used to connect two other elements. It shows the direction that information flows between these elements. Figure 2 is an example of the use of an information flow line. Two bit registers A and B are defined by the appearance of the register symbol, and the information is assumed to flow from A to B. The equivalent IDDAP [15] statement would be ": B = A."

FIGURE 2  
Example of Information Flow Connector



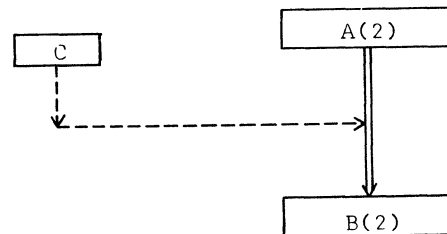
3. Function

The function element allows the user to define a particular set of operations which can be referenced in a subroutine-like manner. The operations are defined by IDDAP statements. The function is activated, in phase 2, by the statement : DO name where name is the name of the function block.

4. Control Signal and Control Flow Line

The control signal and control flow line are used to control a data transfer between two elements. The control signal defines the name by which the transfer is referenced and the control flow line points to the transfer to be controlled. Figure 3 shows an example of control signal C controlling the transfer A to B. The statement : DO C, in phase 2, will cause the transfer to take place.

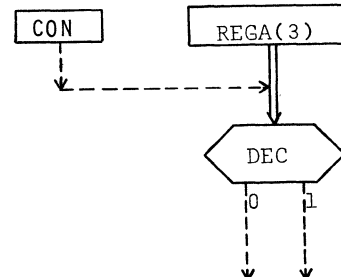
FIGURE 3  
Example of Use of Control



5. Decoder

The decoder decodes an n bit register into one of  $2^n$  control signals. Figure 4 shows a decoder DEC which decodes the register REGA into control signals. Only the decode of zero and one in REGA are shown in the figure. These control signals can be used to control other transfers. The control signal CON controls the decode. When the statement : DO CON is specified, REGA is decoded and the appropriate action takes place based upon the current value of REGA and where the control flow lines point.

FIGURE 4  
Decoder with Control Flow Lines



## 6. Unary and Binary Operand Function Elements

The unary and binary operand function elements allow the user to perform standard functions on the input operands. These operands can be registers or memory. The unary operand function element requires one input operand and the binary operand function element requires two. The result is placed in the register or memory location specified as the output operand. The exact function to be performed is specified by pointing a control flow line at the function name. Tables 2 and 3 specify the functions available with these elements. The use of this feature is best explained by an example. Referring to Figure 5, register A is both the input and output operand. The control signal is INC, and through the control flow lines, it points to the CU portion of the unary operand function. The mnemonic CU means Count Up. When INC is referenced, the result is to add one to the input operand, A, and transfer the result to the output operand, A. In effect, register A is incremented by one. To reference this function, the statement : DO INC causes the A register to be incremented.

TABLE 2

Functions of Unary Operand Function Element

<u>Mnemonic</u>	<u>Meaning</u>
NOT	Logical Not
COM	Two's Complement
LS	Left Shift one position
RS	Right Shift one position
LC	Left Circulate one position
RC	Right Circulate one position
CU	Count Up one
CD	Count Down one

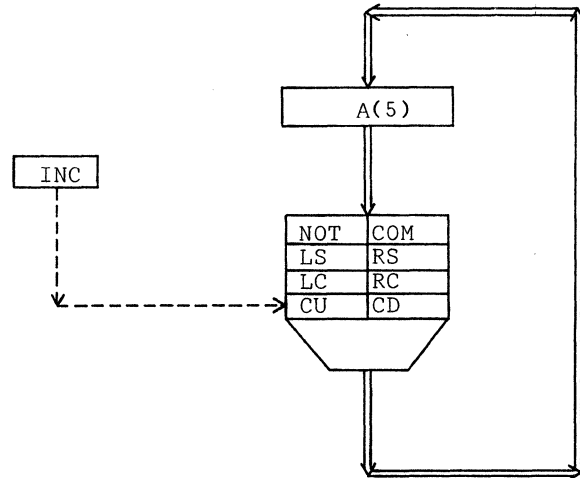
TABLE 3

Functions of Binary Operand Function Element

<u>Mnemonic</u>	<u>Meaning</u>
ADD	Add operand 1 to operand 2
SUB	Subtract operand 2 from operand 1
MUL	Multiply operand 1 by operand 2
DIV	Divide operand 1 by operand 2
REM	Remainder, operand 1 modulo operand 2
OR	Logical inclusive OR of operand 1 and operand 2
AND	Logical AND of operand 1 and operand 2
XOR	Logical exclusive OR of operand 1 and operand 2

FIGURE 5

Example of a Unary Operand Function



## 7. Clock

Simulation has two modes: clock and no clock. In clock mode, update of registers under clock control does not occur until there is a clock pulse, i.e., when the clock variable changes state. In no clock mode, register update takes place immediately [15]. The clock element is used to define the register to be used for a clock.

## 8. Terminal and Subregister

The terminal and subregister allow the user to define terminals and subregisters. Terminals allow the user to refer to a boolean expression by a single name. Similarly subregisters allow the user to refer to a part of a register by a single name. Therefore the text associated with these elements are assignment type statements.

### B. PHASE 2 OR CONTROL FLOW PHASE ELEMENTS

Phase 2 or the control flow phase describes the sequential nature of a digital system in terms of a flow chart. There are only nine basic elements needed in this phase. Three of these tell the simulator where to start and end, and the other six describe functionally the operation of the system.

#### 1. Function Block

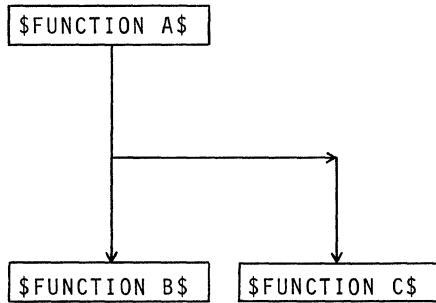
The function block is used to describe a particular function which may be one or more IDDAP statements. It is the basic element of FLOWWARE phase 2. The distinction between this function block and the one in phase 1 is the method of activation. The phase 1 function requires a subroutine-like call whereas the phase 2 function block is activated when it is encountered during the normal sequence of events as specified by the flow chart. In fact, a statement within the phase 2 function block is necessary to call the phase 1 function.

## 2. Go-To Line

The go-to line defines the direction of control flow or the sequence in which operations are to be executed. The order in which elements are executed is determined by the direction of the arrow. Also the user can specify parallel paths with this element as shown in Figure 6. Functions B and C are executed in parallel, after function A has completed execution.

FIGURE 6

Parallel Execution of Function B and Function C  
(Dollar signs \$ denote comments within element)

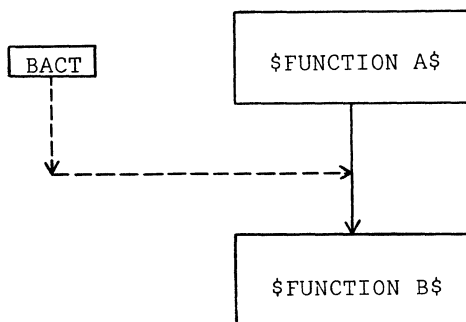


## 3. Control Signal and Control Flow Line

The use of control signals and control flow lines in phase 2 are different from phase 1. It is best explained by the example shown in Figure 7. Function A and Function B are two user defined functions and BACT is a control signal on Function B. Function B is executed only after Function A completes execution and if BACT is true or a logic 1. If BACT is false, the system "waits" until BACT becomes true. If this is the only path in the system and BACT is false, then the simulation will be halted without executing Function B. If there are parallel paths, Function B is executed when BACT is set to logic 1 by one of the other paths.

FIGURE 7

Example of the Use of a Control Signal  
(Dollar signs \$ denote comments within element)



## 4. Decision Block

The decision block is used to decide between two alternate paths. The decision block tests either a boolean expression or a relationship expression, such as  $A > B$ , associated with the element or a control signal pointing to the element. When the expression or control signal evaluates to a logical one, then the exit point is either the top or bottom point of the diamond. If it evaluates to a logical zero, then the exit point is

the right or left point. The exit point is the path to be taken when the decision is made.

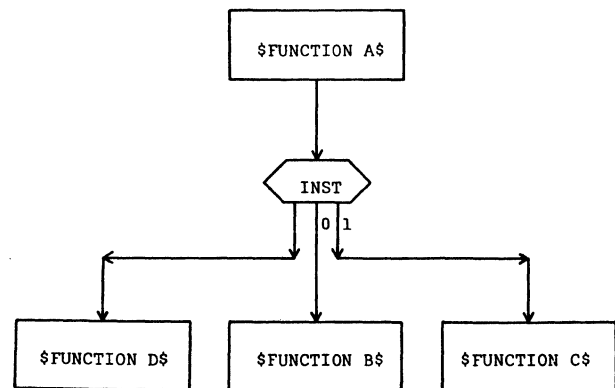
## 5. Decode Block

The decode block decodes the register defined within the block into one of  $2^n$  signals where  $n$  is the length in bits of the register. The go-to lines leaving this element point to the next path to be taken based upon the value decoded. The go-to lines have associated with them a number which represents the decode of the block. A go-to line without a number, only one is allowed, means that for any decoded values not specifically mentioned on other lines, "take this path."

Figure 8 shows an example of the use of the decode block. When Function A completes execution, the register INST is decoded. If  $INST = 0$  then Function B is executed. If  $INST = 1$  then Function C is executed. If anything else, Function D is executed.

FIGURE 8

Example of the Use of the Decode Block  
(Dollar signs \$ denote comments)



## 6. Start, End, and Terminate Blocks

These three elements control the simulation. The simulation starts at the start block and ends at the terminate block. The end block is used to specify the end of a path. When it is encountered, the simulation is not halted but any other parallel paths are executed. The terminate block halts the simulation when it is executed even if there are unexecuted parallel paths.

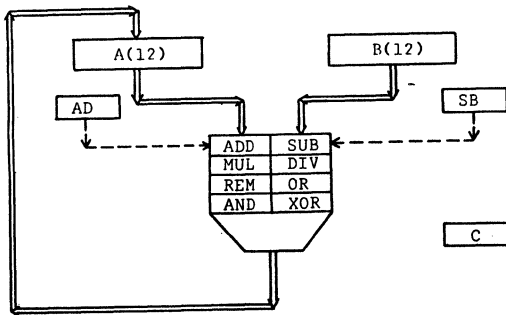
## III. USE OF FLOWWARE

This section will present some simple examples using FLOWWARE. The major emphasis will be on the input language rather than the output of the simulator. As already mentioned, Figure 1 is an example of an up/down counter.

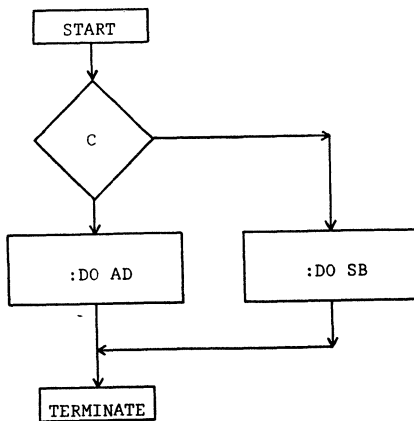
Consider the digital network shown in Figure 9. The problem is to add register A to B or to subtract B from A. In both cases the result is to be transferred to register A. The control signal C is to be used to determine whether addition or subtraction is to be performed. If C is true, perform the addition. Figure 9(a) shows the register layout with signals AD and SB controlling the addition and subtraction respectively. Figure 9(b) shows the control flow where the signal C is tested.

FIGURE 9

Addition/Subtraction Network



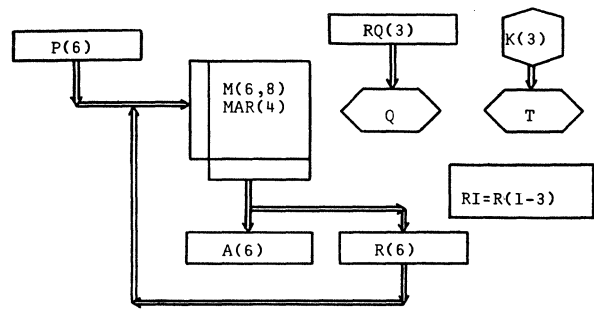
(a) Register Layout Phase



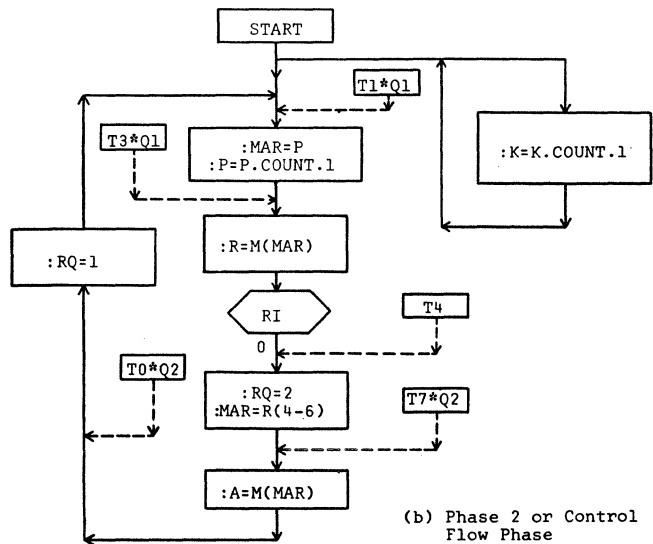
(b) Control Flow Phase

FIGURE 10

Fetch and Execution of an Instruction



(a) Phase 1 or Information Flow Phase



(b) Phase 2 or Control Flow Phase

In Figure 10, the fetch and execution of a LOAD ACCUMULATOR (register A) instruction for a small computer is shown. It is assumed that direct addressing is used and that the six bit computer word is divided in half with three bits for the operation code and three bits for the address. Register P is the program counter, R is the instruction decode register, RI is the operation code subregister, T is the decode of the clock counter K, and Q is the decode of a control register RQ. When RQ = 1, the computer is in the instruction fetch cycle. When RQ = 2, it is in the instruction execution cycle. The operation code for the LOAD ACCUMULATOR is zero. The register layout is shown in Figure 10(a) and the control flow is shown in Figure 10(b). Notice the use of the decode block in both phases as well as the use of the control signals.

IV. CONCLUSION

FLOWWARE has been implemented on the computer system at the University of Missouri-Rolla. This computer system consists of the IBM System/360 Model 50 Computer and several Data General Corporation NOVA-800 Minicomputers. The graphics terminal used as the main input/output device is the T4002 Tektronix Graphic Computer Terminal. FLOWWARE essentially consists of two programs: one written in PL/1 for the System/360 computer and one written in assembler for the NOVA computer. The minicomputer is responsible for drawing the elements and local editing functions, and the main computer is responsible for translation and simulation of the description.

FLOWWARE has been designed for user convenience. The user is relieved of the burdens of drawing the elements and typing long command lines when single letters will suffice. The interactive nature of FLOWWARE permits the user to obtain his results immediately. All elements and all interconnections between elements are clearly visible. Simulation allows the user to see the network "work" under a variety of input conditions. Text and element editing permits modifications to the description. The information flow phase description allows the user to specify graphically a network of registers, etc., which resemble a subroutine and is executed like a subroutine in the control flow phase.

By means of its implementation on the UMR computer system, FLOWWARE has shown itself to be a useful tool

in the process of digital design. FLOWWARE has the flexibility needed to meet a diversity of user demands while still retaining the structural ordering necessary to insure logical consistency within any one description. Its similarity to flow charting, its pictorial nature, its ease of use, and its interactive qualities combine to produce a language which solves the problems present in most text oriented languages, the problems of comprehension and readability.

This project is supported, in part, by NSF Grant GK34076. At present, work is being done on FLOWWARE to improve and expand its capabilities.

#### BIBLIOGRAPHY

1. M. A. Breuer, "Recent Developments in the Automated Design and Analysis of Digital Systems," Proceedings of the IEEE, Vol. 60, No. 1, pp. 12-27, January 1972.
2. C. G. Bell and A. Newell, Computer Structures: Readings and Examples, New York: McGraw-Hill Book Company, 1971.
3. C. G. Bell and A. Newell, "The PMS and ISP Descriptive Systems for Computer Structures," Proceedings of the 1970 Spring Joint Computer Conference, pp. 351-374, 1970.
4. K. E. Iverson, A Programming Language, New York: John Wiley and Sons, 1962.
5. K. E. Iverson, "A Programming Language," Proceedings of the 1962 Spring Joint Computer Conference, pp. 345-351, 1962.
6. K. E. Iverson, "A Common Language for Hardware, Software, and Applications," Proceedings of the 1962 Fall Joint Computer Conference, pp. 121-129, 1962.
7. H. Schorr, "Computer-Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions on Electronic Computers, Vol. EC-13, pp. 730-737, December 1964.
8. J. R. Duley and D. L. Dietmeyer, "A Digital System Design Language (DDL)," IEEE Transactions on Electronic Computers, Vol. C-17, pp. 850-861, September 1968.
9. J. R. Duley and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Transactions on Electronic Computers, Vol. C-18, pp. 305-313, April 1969.
10. Y. Chu, "An ALGOL-Like Computer Design Language," Communications of the ACM, Vol. 8, pp. 607-615, October 1965.
11. T. C. Bartee, I. L. Lebow and I. S. Reed, Theory and Design of Digital Machines, New York: McGraw-Hill Book Company, 1962.
12. D. L. Parnas, "A Language for Describing the Functions of Synchronous Systems," Communications of the ACM, Vol. 9, No. 2, February 1966.
13. D. L. Parnas, "More on Simulation Language and Design Methodology for Computer Systems," Proceedings of the 1969 Spring Joint Computer Conference, pp. 739-743, 1969.
14. J. A. Darringer, "The Description, Simulation, and Automatic Implementation of Digital Computer Processors," Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1969.
15. R. F. Crall, "IDDAP--Interactive Computer Assistance for Creative Digital Design," Ph.D. Dissertation, University of Missouri-Rolla, Rolla, Missouri, 1970.
16. K. E. Iverson, A. D. Folkoff and E. H. Sussenguth, "A Formal Description of System/360," IBM Systems Journal, Vol. 3, No. 3, pp. 198-262, 1964.
17. D. M. Rouse, "A Design Oriented Digital Design Language," M. S. Thesis, University of Missouri-Rolla, Rolla, Missouri, 1969.
18. C. G. Bell, J. L. Eggert, J. Grason and P. Williams, "The Description and Use of Register-Transfer Modules (RTM's)," IEEE Transactions on Computers, Vol. C-21, No. 5, May 1972.
19. C. G. Bell and J. Grason, "The Register Transfer Module Design Concept," Computer Design, May 1971.
20. Advertisement from Digital Equipment Corporation, Maynard, Massachusetts, 1972.
21. J. L. Brame and C. U. Ramamoorthy, "An Interactive Simulator Generating System for Small Computers," Proceedings of the 1971 Spring Joint Computer Conference, pp. 425-449, 1971.
22. W. E. Omohundro, "FLOWWARE--A Flow Charting Method to Describe Digital Systems," Ph.D. Dissertation, University of Missouri-Rolla, Rolla, Missouri, 1973.
23. W. E. Omohundro, "FLOWWARE Users Manual," Technical Report, University of Missouri-Rolla, Rolla, Missouri, 1973.
24. W. E. Omohundro, "FLOWWARE Implementation Package," Technical Report, University of Missouri-Rolla, Rolla, Missouri, 1973.



# AUTOMATED EXPLORATION OF THE DESIGN SPACE FOR REGISTER TRANSFER (RT) SYSTEMS

Mario R. Barbacci  
Daniel P. Siewiorek

*Departments of Computer Science and Electrical Engineering  
Carnegie-Mellon University*

**KEYWORDS AND PHRASES:** Design Automation, Register Transfer Level, Design Space, Cost/Speed Trade-offs, Register Transfer Modules.

**ABSTRACT.**— A Design Automation System for the RT level of design is described. The System explores the design space by finding alternative implementations for a user given behavioral specification. The alternative solutions are obtained by transformations on a graph model. These transformations effect trade-offs between the cost of the hardware and the speed of the algorithm. Heuristic routines are used to reduce the design space by exploring only those alternatives whose characteristics approach a user given set of goals.

## 1. INTRODUCTION

A computer system is composed of thousands of interconnected components. The basic components of computer systems have gone through an evolution from relays, to vacuum tubes, to transistors, to logic gates (small scale integration), to registers (medium scale integration), and to memories and processors (large scale integration). As the basic components increased in logical power more complex computer systems became feasible.

The construction of these computer systems has been simplified by computer aided design. Early attempts at design automation were directed towards a reduction in cost and time of the design process itself [1]. These objectives were accomplished by relieving engineers of repetitive time consuming tasks. This approach to design automation limits itself to filling the gap between the low level design specifications and the manufacturing data. The inputs to the systems are, generally, in terms of Boolean equations which the system then translates into an equivalent gate level specification. The Boolean equations specify the desired behavior of the finished object. Most of the synthesis algorithms at this level deal with the problem of reduction or simplification of the Boolean equations.

Recent efforts at design automation have been directed towards a system capable of accepting a high level description and translating it into an equivalent gate level structure. APDL[2] and ALERT[3] are two such systems.

The essential feature lacking in these existing systems is the exploitation of alternative implementations derived from the initial behavioral specifications. This paper deals with the description of an automatic design system that explores the design space for the register transfer level. The Register Transfer (RT) level [4] is characterized by the following basic components: Registers, register transfers, and transformations on the contents of registers. When completed, the system will take as inputs the specification of the desired behavior in some high level RT language and the specifications of the hardware RT level components. The output is the specification of the hardware which attempts to optimize the system along some specified dimensions of the design space. We will restrict ourselves to the

-----  
The research in this paper was supported by National Science Foundation Grant GJ 32758X.

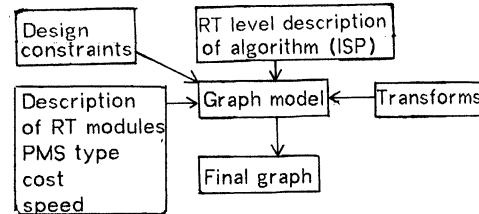


Figure 1. An RT level design automation system

cost and time dimensions. Thus a designer specifies design constraints to the system, such as whether the solution should be the cheapest, the fastest, or some trade-off between cost and speed.

The automatic design system is depicted schematically in Fig. 1. The description of the algorithm is given in the RT language ISP [4] and translated into a graph representation. The user can, however, bypass this step and provide its input — the graph — directly to the System in an assembly-like notation. This can be used to design systems not describable in ISP. Subsequently, various transforms on the graph are attempted to establish a new solution to the problem. A set of heuristics guide this exploration of the design space by using the given design constraints to decide which solutions should be kept to generate other solutions by yet another application of the graph transformations.

Which set of transforms to apply is determined by the PMS (Processor Memory, Switch) type [4] of the modules. The set of transforms are general and can be used with any set of modules which conform to a particular PMS type. Transforms (module dependent) which depend on the details of a certain module set, such as the cost/performance ratio between two modules of the same type, are not included in the general (module independent) set of transformations although it is a simple task for a designer to add extra transforms to the set.

The following sections describe different portions of the system. Sections 2 and 3 describe the system inputs, the module set and the initial description of the algorithm to be implemented. Section 4 delineates the PMS types which are used to select the set of transforms discussed in section 5. The cost or gain achieved by applying a transform is treated in section 6, while the heuristics which drive the design process are presented in section 7. Finally, an example problem is given in section 8. The various sections will be treated by way of examples. The complete details can be found in [5].

## 2. A MODULE SET

To lend credibility to the discussion of the system, a commercially available [6] set of RT level modules, called Register Transfer Modules (RTMs), will be used as the module set.

The following paragraphs briefly introduces the modules and discusses the design process using them. A more detailed description is given in [7]. The flowchart format of the RTM notation is so transparent, however, that the detailed reference probably need not be read to understand this one.

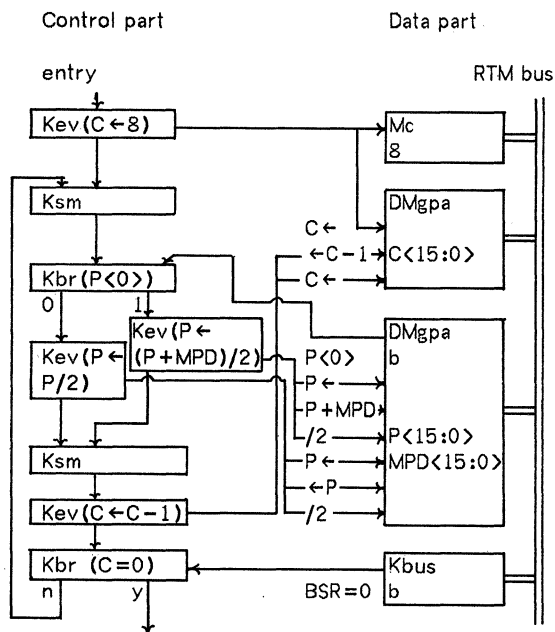


Figure 2. An RTM multiplier

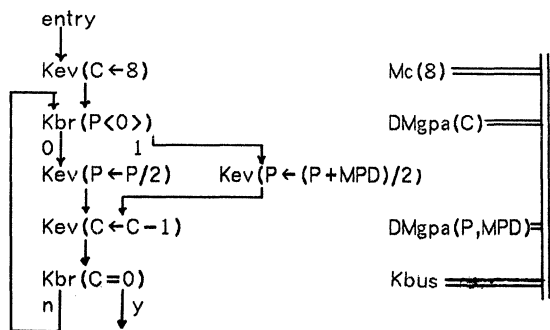


Figure 3. Short hand notation of the RTM multiplier

The RTM set consists of about 35 module types falling into four classes. Each RTM system is built around a common bus for facilitating data transfers among the registers of the modules connected to it. The three types of modules that connect to the bus are: M's - Memories for holding single bits (Boolean), or 8-, 12-, or 16-bit integers, and arrays for holding vectors of integers; T's - transducers for interfacing with the environment external to RTM (e.g. lights and switches, analog-digital converters, serial interfaces for teletypes); and DM's - Data-Memory components to hold data and carry out logical and arithmetic operations on this data. A fourth type of module, the K-type, controls the operations in the other three. A network of K modules is isomorphic to the flowchart of the computational algorithm that is to be performed, and each individual K module evokes some operation(s) in the data part of the system (centered around the bus). The bus has timing interlock signals to interlock data transfer operations evoked by the K modules. Multiple buses can be used to increase the performance of a system.

Figure 2 depicts the RTM implementation of an 8-bit shift and add multiplier and Fig. 3 the short hand notation for the

```

Multiplier := (C ← 8; next
Loop := ( (P < 0) ⇒ P ← (P + MPD) / 2;
          (~P < 0) ⇒ P ← P / 2; next
          C ← C - 1; next
          (C ≠ 0 ⇒ Loop) )
);

```

Figure 4. The ISP description of the RTM multiplier

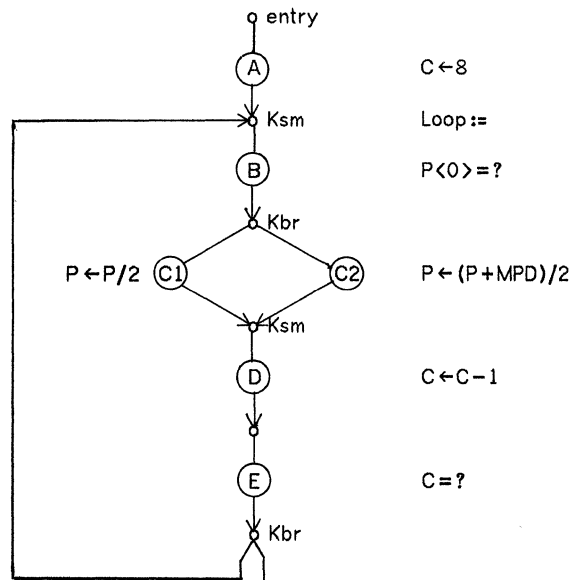


Figure 5. The graph model of the multiplier

system. The multiplier is in the P register and the multiplicand is in the MDP register and is assumed to occupy the leftmost 8 bits of the register. The product will be in the P register. The partial products are formed in the left hand side of the P register and shifted to their appropriate position in the final product after eight transferences of the loop. The multiplier will be used as an example for the following discussion.

### 3. THE GRAPH MODEL

There are five basic types of operations in the graph model the design automation system uses:

- branch (Kb), activates one of the output paths depending on Boolean conditions
- serial merge (Ksm), activates its output path when any of the input signals arrive
- diverge (Kdiv), activates concurrently all paths attached to it
- parallel merge (Kpm), activates its output path when all its input signals have arrived
- data operations (other)

The translation process from the input RT language description (ISP) to the graph model is straightforward and has been programmed. The ISP for the multiplier is shown in Fig. 4 and the corresponding graph model is depicted in Fig. 5.

The system as implemented treats each node in the graph as composed of a non-empty sequence of the five operation types. The only restriction is that nodes must have a unique entry operation (Ksm, Kpm, or data operation) and a unique exit operation (Kb, Kdiv, or data operation). In the examples that follow, we will explicitly show the control operations by drawing them outside their nodes.



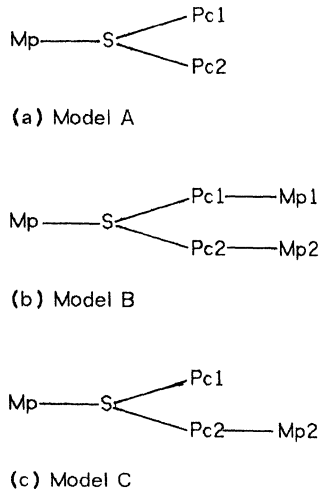


Figure 6. PMS types

#### 4. PMS MODULE TYPES

The decision as to which set of transformations should be used is determined by which PMS types the module set can emulate.

In model A (Fig. 6.a), each process communicates directly with a single large main memory. The important feature is that each process can modify information which is to be used by the others.

In model B (Fig. 6.b), slave memories (buffers) have been added to the system. A process can fetch information from main memory, but any information to be stored is put in its buffer. The buffer acts as intermediate storage between the process and the main memory. When a process needs some information it looks first in the associated buffer to see if the information has been stored there as a result of a previous computation. If not, the data is obtained directly from main memory. When both processes have completed their tasks, the information in the slave memories is transferred to the appropriate locations in main memory.

Model C (Fig. 6.c) differs from model B in that only one slave memory is used. One of the processes (Pc1) can fetch and modify data directly in the main memory. The other process (Pc2) can only fetch data from main memory and uses Mp2 as a buffer for partial computations.

From these models the various conditions on the variables for parallel processing can be developed [8,5]. RTM's correspond to either model B or C since a process occupies a bus and two busses cannot share data without co-operation between processes.

#### 5. THE TRANSFORMS

The set of transforms for RTM's will be demonstrated by example. The full set of transforms is described elsewhere [5]. In general, speed is achieved (at some extra cost) by increasing parallelism. Cost is decreased by reducing parallelism. For purposes of example, suppose that we want to increase speed.

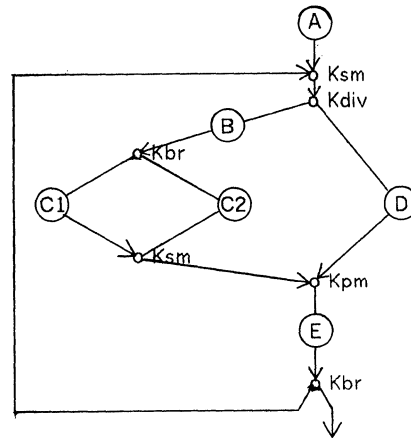


Figure 7. The parallel computation of nodes D and B,C1,C2

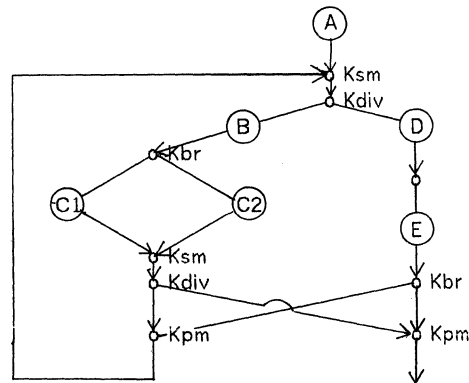


Figure 8. The parallel computation of nodes D,E and B,C1,C2

Consider the multiplier in Fig. 5. The graph model is first cleaned up by removing no-operation control nodes (Kpm with a single input for example) which were introduced by the ISP to graph model translation.

Associated with each node is a, possibly empty, set of variables which indicates which variables are used and/or modified by the operation(s). Node D ( $C \leftarrow C-1$ ) depends on variable C alone while nodes B, C1, and C2 depend on P and MPD. Hence node D can be computed in parallel with nodes B, C1, and C2 since they depend on different sets of variables. This is depicted by the transformed graph in Fig. 7. Note further that node E also depends only on variable C. Hence E could be performed in parallel with B, C1, and C2, but it must follow the computation of D, as shown in Fig. 8.

Sometimes one node may use a variable while another uses and modifies the same variable. The first node can be computed in parallel with the second if the first node receives its own copy of the variable before the parallel computations starts. Copying the variable takes time and requires extra hardware. By defining the various ways variables are used it is possible to determine if a transformation can be applied and how much will be saved or lost in terms of time and cost as shown in the next section.

The transforms are of a general nature in that they apply not only to individual nodes but to subgraphs of arbitrary complexity. Each subgraph is also characterized by the variables used and/or modified by its computations. Methods for forming these subgraphs and their associated variable sets have been automated [5] but will not be described here.

## 6. DESIGN SPACE TRADE-OFFS

Two parameters will be used to describe the design space: The cost of the hardware involved and the operational time. The former is obtained by adding the costs of the components used in both the data and control structures. The latter is obtained from the average speed of the operations involved.

For a straight sequence of operations the time required is the sum of the individual times, Fig. 9.a. In the presence of concurrent activities, the operation time is that of the longest (timewise) sequence, Fig. 9.b. When alternative sequences are initiated as a result of a data dependent decision, the time required for the execution is not known a priori. In this instance a worst case situation will be assumed, namely, that the longest path is the one selected, Fig. 9.c.

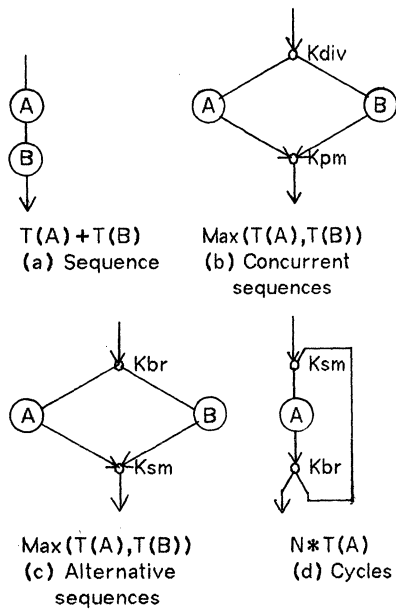


Figure 9. Time estimation

The presence of cycles (loops) adds some complexity to the estimation of the operation time. In this case the level of nesting is assumed to be proportional to the frequency of execution of the operations. Conceptually this is equivalent to replacing the cycle by a sequence of multiple copies of the individual operations. Since the number of times a loop is executed (i.e. the number of copies) is usually unknown, a default (2) is assumed. This default may be overruled by the designer by specifying an estimate loop count. Fig. 9.d.

Having defined the parameters of the design space we can now describe the trade-offs involved in the transformation rules. Connectivity and data dependency are used in the system to indicate the feasibility of a transformation. Feasible transformations, however, do not imply necessarily any advantage in their application, and the desirability of such a transformation is indicated by a different set of conditions.

Fig. 10 shows the effect of one of the transformations, rule SP. Node X1 is required to copy to local memory those variables used by node (subgraph) B in its computation according to PMS types B and C. Likewise the two X2 nodes are required so that all the variables transformed by nodes A and B are available to

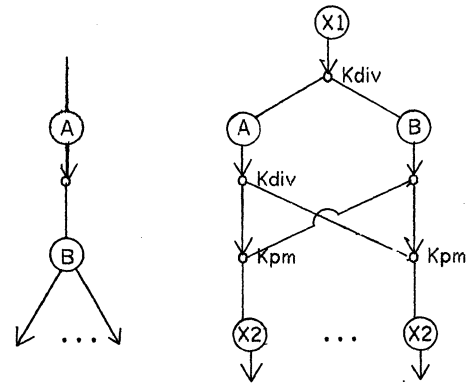


Figure 10. Rule Serial to Parallel (SP)

any of the  $n$  paths originally following node B. The trade-offs are:

TIME:	Original	$T(A) + T(B)$
	new	$T(X1) + T(X2) + \text{MAX}(T(A), T(B))$
	gain	$T(A) + T(B) - T(X1) - T(X2) - \text{MAX}(T(A), T(B))$
COST:	Original	$C(A) + C(B)$
	new	$C(X1) + n.C(X2) + C(A) + C(B) + \alpha.C(\text{Bus})$
	extra	$C(X1) + n.C(X2) + \alpha.C(\text{Bus})$

Where  $\alpha = 0$  or 1, depending on the availability (e.g. idle) in the current version of the system of a bus that could be used by B.

In the case of rule SP the concurrent computation of A and B may not bring about a reduction in time: The transfer operations X1 and X2, used to load and unload variables to and from the different busses, take a non-zero amount of time. If the number of variables transferred is large, this overhead may cancel any gains obtained from the concurrent computation of A and B. The bus required to execute B may or may not be already present in the system. If it is available ( $\alpha = 0$ ) then it can be shared at no extra cost.

Desirability conditions for other transformations are described in [5]. They can be used to eliminate those (feasible) transformations where they do not produce the desired savings (in cost or time) or where the gain is below a designer specified threshold.

## 7. HEURISTICS

Due to the interaction between transformations it is a difficult task to formalize the optimization (improvement of alternative structures) as a mathematical optimization problem. The main difficulty is the fact that transformations apply to subgraphs of arbitrary size, and as a consequence transformations in a given alternative structure may or may not be feasible or desirable in structures derived from it. It is also the case that new cases of transformations become feasible or desirable only after a specific sequence of transformations has been applied.

The design space is represented by a time/cost diagram. Alternative structures are represented by points in the diagram. Except for the original solution, all points are derived, by transformations, from other points in the space. These relationships will be made explicit by drawing vectors from the parent nodes to their immediate (i.e. one transform removed) descendants.

The exploration of the design space in our system is performed by a group of heuristic routines that produce alternative designs in a goal oriented fashion, the goal being specified by the designer. Ideally, the goal is to find an alternative structure whose position in the design space is as close as possible to the origin (0 cost and 0 time). This ideal case is, however, not easily found in real solutions. The usual case is that the least expensive solution is not the fastest and vice versa. This characteristic provides a rough classification of the design objectives into two classes: minimal cost and minimal time.

Although a designer's aim can be classified according to these objective functions it may be the case that the real objective is more complicated in nature, namely, some combination of time and cost. For instance, the objective could be something like: "the fastest alternative structure not costing more than x dollars".

For simplicity, the subspace of acceptable solutions will be defined by a set of straight line segments whose slopes reflect the objective functions. In the example above a single straight line, parallel to the cost axis would be used to divide the space in two halves. Only those solutions that lie in the semispace containing the origin are considered acceptable. These solutions

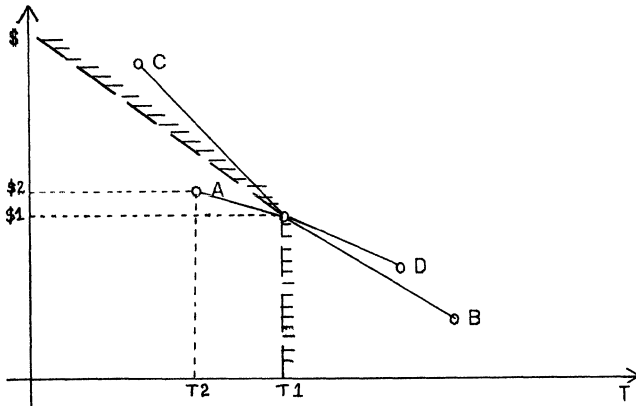


Figure 11. Design space reduction

represent improvements along the design goal.

More complex constraints can be described by using lines of the form  $\$ = -m.T + b$ , where  $m$  is a parameter indicating how many dollars the designer is willing to pay for each time unit saved (if time is the primary goal) or how many time units the designer is willing to sacrifice for each dollar saved (if cost is the objective). An example, Fig. 11, will clarify this description.

Assume that the primary objective is a reduction in time, and that the designer wants a time/cost trade-off of at most  $m$  dollars for each time unit improvement. Furthermore, assume that the original design is characterized by  $\$1$  and  $T1$ . The "acceptable trade-off" subspace would thus be delineated by two line segments: one parallel to the cost axis starting from  $(T1, \$1)$  to  $(T1, 0)$ , and the other through  $(T1, \$1)$  with slope  $-m$ . By studying the control flow and data dependencies in this original structure, four transformations are available which yield four alternative solutions derived from the original one: A, B, C, D.

By dividing the space according to the trade-off lines, alternatives B, C, and D can be rejected because their characteristics are not within the acceptable subspace (i.e. they take more time or the decrease in time costs too much). The alternative left, A, represents improvement in time while the cost to achieve the improvement is under the designer's threshold.

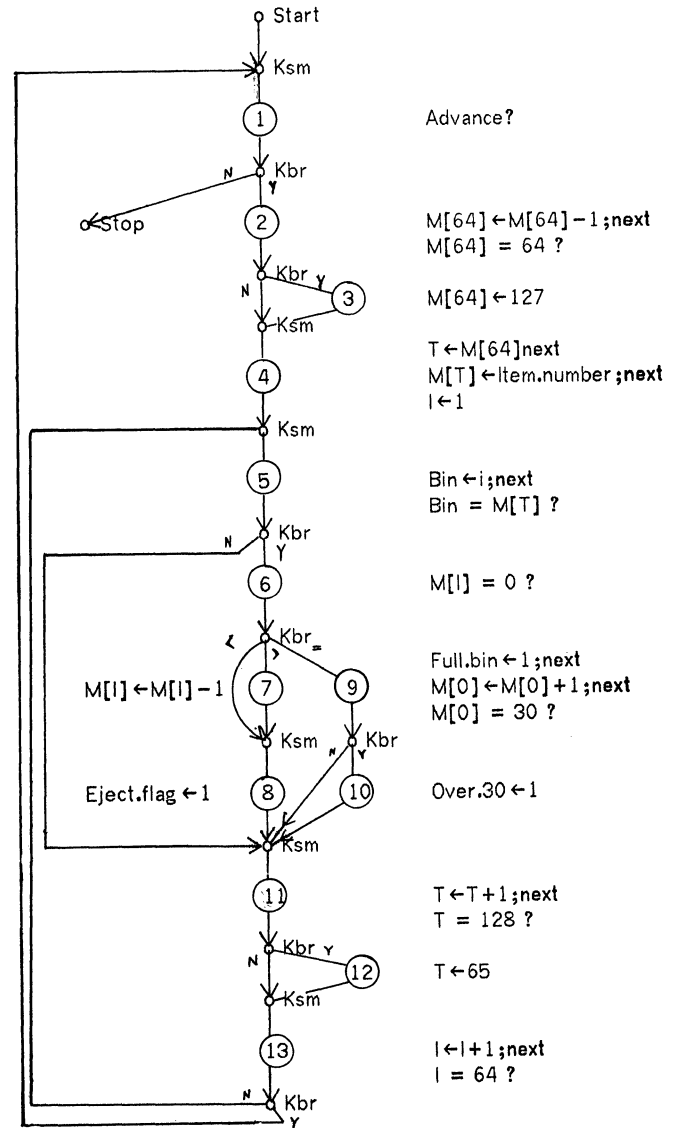


Figure 12. Controller for a conveyor-bin system

The process can now be applied to A in an identical manner. Design A is taken as the new initial solution and a new "acceptable trade-off" subspace is defined by a line segment  $(T2, \$2)$  to  $(T2, 0)$  and a line with slope  $-m$  through  $(T2, \$2)$ . Since in some cases more than one alternative can be left for further exploration, this process takes the form of a tree walk where the nodes represent alternative solutions and the edges are the transformations applied. In some instances, identical structures can be obtained by different sequences of transformations and the exploration of the design space is a graph walking process. In any event, a path ends when no alternative solutions worth exploring can be reached from a given point. When all possible paths have been explored the end nodes are measured against the primary objective and the best one chosen.

#### 8. A CONTROLLER FOR A CONVEYOR-BIN SYSTEM

The following example is taken from [7]. Briefly, the algorithm performs the controlling function for a conveyor carrying items to be sorted into bins.

The algorithm is described in ISP and its graph model is shown in Fig. 12. Notice that in this example the nodes correspond to sequences of one or more operations.

## 9. CONCLUSIONS

The purpose of this paper is to describe the development of an automated method for designing digital systems at the RT level. The designed system is optimized along a set of designer constraints. The primary result is a system that translates an initial behavioral description of a digital system into alternative structural specifications from which it can be built. For simplicity, the structural specifications are given in terms of a specific set of building blocks, the RTM set.

Due to space limitations, it is impossible to provide in a paper of this nature any detailed description of the system as implemented, and therefore we have tried to point out in general terms what its capabilities are.

The system is a research tool and its implementation allows it to be used either as a closed system, in which the user only specifies an initial description and a set of constraints and goals, upon which the system performs an automatic design space exploration; or, as an interactive facility, driven by a command language that allows the user to exercise any function of the system from a time-sharing terminal.

A system of this nature presents limitations as to the degree of "optimization" it can perform. It is not expected to obtain solutions that are radically different from the one specified by the user. Hence, its use is more likely to be as part of a design cycle, in which the user presents an initial description which is processed by the system; the result of this is an exploration of the design space around such initial solution; this exploration can suggest to the user modifications to his behavioral specifications; this modified specifications are then fed back into the system and the process starts again.

## REFERENCES

- [1] Breuer, M.A.: "Recent developments in the Automated Design and Analysis of Digital Systems". IEEE Proceedings, Vol. 60, No. 1, January 1972, pp. 12:27.
- [2] Darringer, J.A.: "The Description, Simulation, and Automatic Implementation of Digital Computer Processors". PhD thesis, EE Department, Carnegie-Mellon University, May 1969.
- [3] Friedman, T.D. and Yang, S.: "Methods used in the Automatic Logic Design Generator (ALERT)". IEEE-TC, Vol. C-18, No. 7, July 1969, pp. 593:614.
- [4] Bell, C.G. and Newell, A.: "Computer Structures: Readings and Examples". McGraw Hill Book Company, New York, 1971.
- [5] Barbacci, M.R.: "A Register Transfer Automatic Design System". PhD thesis, CS Department, Carnegie-Mellon University, December 1973.
- [6] Digital Equipment Corporation: "PDP16 Computer Design Handbook". 1971.
- [7] Bell, C.G., Grason, J., and Newell, A.: "Designing Computers and Digital Systems". Digital Press, Digital Equipment Corporation, 1972.
- [8] Bernstein, A.J.: "Analysis of Programs for Parallel Processing". IEEE-TEC, Vol. EC-15, No. 5, October 1966, pp. 757:763.

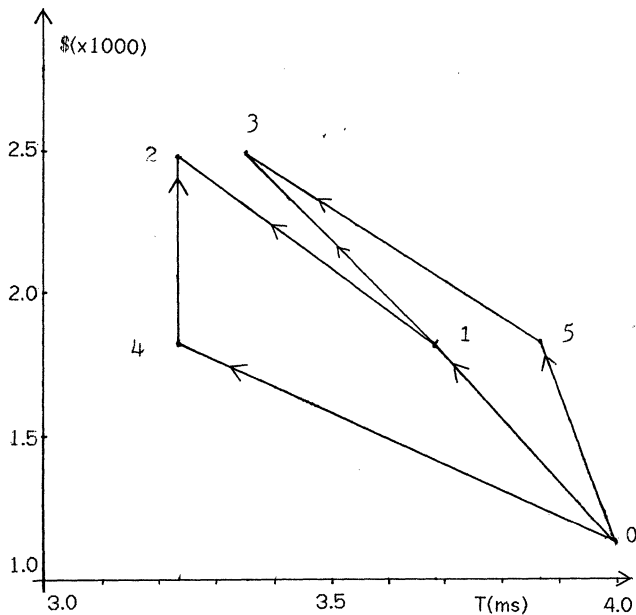


Figure 13. Design space exploration

Several alternative implementations can be derived from this example. They are rather simplistic due to the compactness of the algorithm, but they are nevertheless appropriate to show the design space and its exploration, Fig. 13. First, assume that the fastest solution is sought. All the applicable transformations deal with the increment and testing of variables T and I (nodes 11,12, and 13 of the flowchart), and their concurrent execution with the main computation (nodes 5,6,7,8,9, and 10).

The best solution (timewise) is given by point 4 in the design space. In this solution, the main body of the algorithm (5,6,7,8,9,10) is computed in parallel with the increment and testing subprocess (11,12,13) as a whole. Other alternative points are also shown in the diagram (points 1,2,3,5,6). Several things can be noticed in the design space diagram; for instance, point 2, the parallel computation of (5,6,7,8,9,10), (11,12), and (13) is reached in two ways: First, (5,6,7,8,9,10,11,12) is performed in parallel with (13), point 1, and then the larger computation is performed as (5,6,7,8,9,10) in parallel with (11,12). The other way of reaching point 2 is by computing (5,6,7,8,9,10) in parallel with (11,12,13), point 4, and then transforming the smaller subgraph into (11,12) in parallel with (13). Notice furthermore, that points 2 and 4 present the same time value. The system uses the distance to the origin as a tie breaker parameter.

The same example was also processed with the constraints that 1) No more than 3 cost units (dollars) were to be added for each time unit (1 microsecond) of speed-up, and 2) The time should be no greater than the initial solution. With these constraints, the system rejected point 5 for not having the proper trade-off with respect to its predecessor. It is interesting to see that point 3, which could be reached from 1 and 5 under the unlimited cost constraint, can only be reached from 1 (since 5 was rejected, its successors were not obtained). A similar situation is present at point 2, with respect to points 1 and 4. The interesting detail is that, 2, when reached from 1 is accepted since the trade-off involved is below the threshold. When 2 is reached from 4, it is rejected since the trade-off involved is beyond the threshold.

# IMPLEMENTATION ASPECTS OF THE SYMBOL HARDWARE COMPILER

T. A. Laliotis  
Fairchild Systems  
Palo Alto, California

## ABSTRACT

One of the most outstanding features of the SYMBOL computer is its high level hardware compiler. This paper presents some aspects of the hardware implementation including the network characteristics of the communication scheme between compiler, system supervisor, and Memory Controller, the functional breakdown into distinct sections for implementation, the support hardware (registers, tables, etc.), the Name Table structure, and some of the linking techniques for the structured output of the compiler.

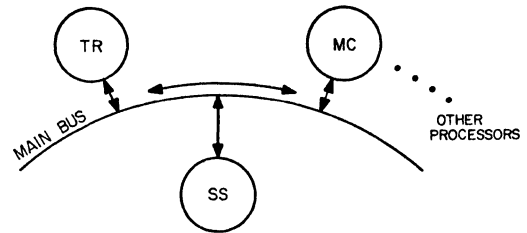


Figure 1. SS, TR, and MC Communication

## I. INTRODUCTION

The main objectives and goals of the SYMBOL research project [1,2,4] were to demonstrate the reduction of the total costs of data processing by revising the designer's approach on the following key items:

- Hardware/Software boundaries
- System Architecture
- System Packaging

The hardware compiler is one of the best examples for demonstrating items (a) and (b) above because, first, it was implemented totally in hardware thus representing a 100% departure from the classical approach of totally software compilers and second, the language used (the "SYMBOL" language) [3] broke all barriers of traditional restrictions for compatibility with existing languages. The SYMBOL SYSTEM consists of the following eight specialized processors which operate autonomously but are linked together via the main data and communication bus: the System Supervisor (SS), the Memory Controller (MC), the Compiler (Translator) (TR), the Central Processor (CP), the Channel Controller (CC), the Input Processor (IP), the Disc Channel Controller (DC), and the Memory Reclaimer (MR).

The Compiler takes as its input a program written in the high level procedural "SYMBOL" language. The program has been deposited in the Memory by the IP. The Compiler then generates a reverse polish object string and a multi-level block structured name table suitable for execution by the Central Processor. In the process of doing this, the Compiler uses a small table of Reserved Words (about 100) which are kept in the non-pageable portion of main memory and a library of call-by-name system procedures stored in the pageable portion of main or bulk memory. The compiler manages its own communications with the Memory Controller and the System Supervisor. All of the above objectives are accomplished totally in hardware.

## II. COMPILER OVERVIEW

Basically, the Compiler can be thought of as a network in conjunction with the System Supervisor (SS) and the Memory Controller (MC). See Figure 1. For reasons of compatibility with previous SYMBOL references, the compiler will hereafter be referred to as the Translator (TR). Each mode of communication will be discussed in detail later.

The overall block diagram of the Translator and its parts of communication with the SS and MC are shown on Figure 2. As indicated there, the TR picks up its input (source program) from some location in storage called TWA (Transient Working Area) and deposits its two structured outputs (Object Code and Program Name Table) in other locations of storage. It also communicates with the SS which maintains the Terminal Control Headers, the Task Assignment Queues, the Page Out Queues, and handles the Error and Interrupt analysis.

From the standpoint of hardware implementation, the TR is divided into three major sections as shown in Figure 2: the Object Code section, the Name Table section, and the Support Hardware section. Only the Name Table and Support Hardware sections will be dealt with in this paper. From the functional standpoint, only one of the three sections can be active at a time. Either one of the two logic sections (Object or Name Table) can request action by the Support Hardware but once the Support section has been activated, the logic section that requested the action freezes until the end of the Support activity at which time it continues on. During compilation, the operation of the two logic sections is a Ping-Pong-like action. The Object section processes all non-literal single characters (delimiters) and structured alphanumeric data until it comes to a blank space followed by a letter; this could be either a Reserved Word or an identifier. At that point it turns control over to the Name Table section. The Name Table section resolves the name and gives control back to the Object section for processing. Thus, the control bounces back and forth between the two sections until the end of the source program. At that time the Name Table section takes over and performs the resolution and linking of all identifiers (Global Linking).

## III. SUPPORT HARDWARE

### A. TR-SS COMMUNICATION

TR-SS Communication takes place during Control Exchange Cycles (CEC). During a CEC, a certain allocation of bus lines is used for communicating information between the eight processors in the system.

The process of compiling a job begins at the end of the Load Mode administered by the IP. At that time the Input Processor (IP) notifies the System Supervisor (SS) during a CEC, that it has finished inputting a program, the SS then puts that program (job) at the bottom of the Translator queue and also initializes the Terminal Header Control Words [5] with the appropriate pointers to the beginning address of the source code and to the beginning address of the object code which is to be

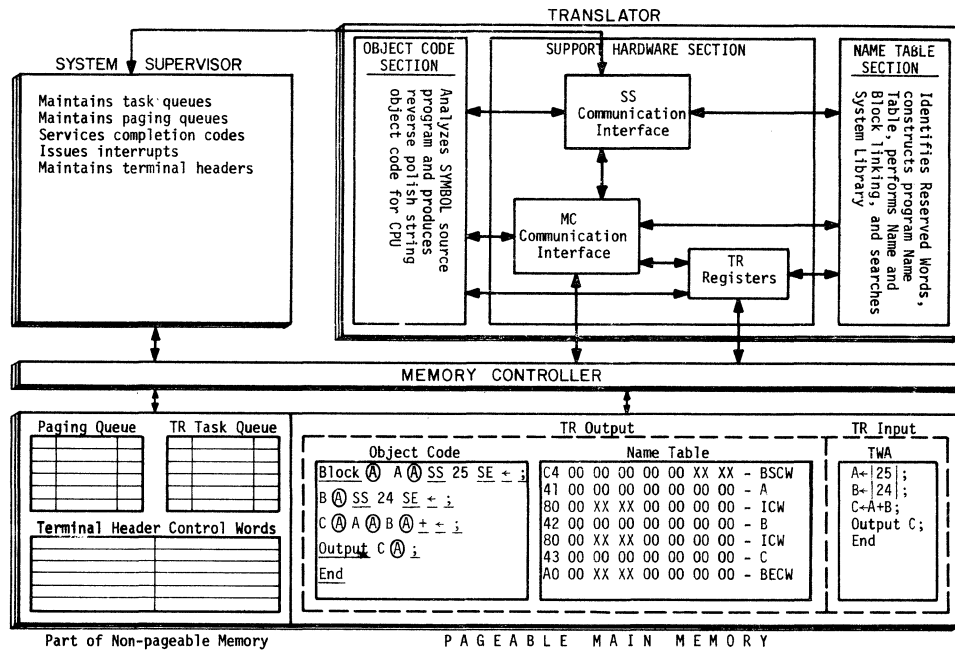


Figure 2. Overall Block Diagram of Translator

generated by the TR.

When the job percolates up to the top of the TR queue, the SS initiates a Control Exchange Cycle and sends a start command to the Translator over the control bus along with the Terminal (user) number of the Terminal that inputted the job.

At this point, the Translator becomes activated, looks at the terminal number, and begins work on the job by first fetching the Terminal Header Control Words to find its pointers. The Translator is now on its own and, from this point on, it can be stopped only by the occurrence of one of the following conditions: SS Interrupt, Program Trap, Page Out, Program Error, and Completion of Task.

In each one of these cases, the TR saves its status in the appropriate terminal header control words, initiates a CEC, and transmits a completion code to the SS during the CEC. The SS analyzes the code and takes the appropriate action. Specifically, in the case of Program Error, the TR saves enough information so that the SS under system control will print out at the user's terminal the type and location of the syntax error. In the case of a Page Out, both SS and TR sense the Page Out from the Memory Controller (MC). The TR starts its shutdown procedure, the SS performs some housekeeping for the TR Page Out but does not wait for the TR completion code. It goes on with whatever other tasks it may have in its queue until a Page Out completion is received from the TR. The page is then put on the paging queue, the Terminal Header Control Work (THCW) of the task is marked to indicate waiting for a page and the job is put on the bottom of the TR queue. Another task is now assigned to the TR. When the page has been brought in by the MC, the THCW is marked to indicate that the page is now in. The next time the task percolates up to the top of the TR queue again, the SS restarts the TR on that job. The TR, during its shutdown process, saves the following information: Name Register, Stack Register, Object Register, All Address Registers, Phase Counter Status, all pertinent flags, and source character pointer.

#### B. TR-MC COMMUNICATION

The SYMBOL system features a Dynamic Memory Management capability via the Memory Controller which allocates memory space on demand, performs address arithmetic, and manages the associative memory needed for paging in its virtual memory environment.

The Translator is one of the heaviest users of Memory. Besides its input-output interaction with memory (fetch source program, store object string, and name table), it performs many searching operations. For every English word that appears in the source program, the Reserved Word Table (RWT) has to be searched. If it is not found in the RWT, the current block of the Name Table has to be searched. At the end of the program the entire Name Table has to be searched again for Global Name resolution and linking, procedure call handling, and system name resolution. Thus, to avoid situations where the TR would tie up the memory during long searches, the TR was given a relatively low memory access priority (fourth priority out of five; after SS, IP, CP, but before MR).

Figure 3 shows the TR-MC communication in a block diagram form. Typically, a phase of a particular Task Phase Counter requests a memory cycle by raising the MOP (0-3) lines and holds at that phase. The Memory Communication Interface Section takes over. It puts out TR's Memory request line (MP4). When the Memory is free and there is no higher priority request, it puts the command on the bus and unloads the registers. A few clocks later, the MC returns a completion code during a CEC. If the Memory operation was successful, the TR loads its Registers from the main bus during the clock time following the CEC and the advance signal (MDONE) is issued to allow the logic phase counter to move on to the next phase.

If a page out completion was returned, the MDONE signal is not issued. Thus, the logic phase counter freezes and the shutdown routine takes over and saves the TR status, including the state of the phase counter.

During the period between cycle granting and completion code return, from the MC, the main bus can be used by the CP on a cycle-stealing basis for intra-CP data communication between the various sub-units of the CP (Instruction Sequencer, Arithmetic Processor, Reference Processor, and Format Processor).

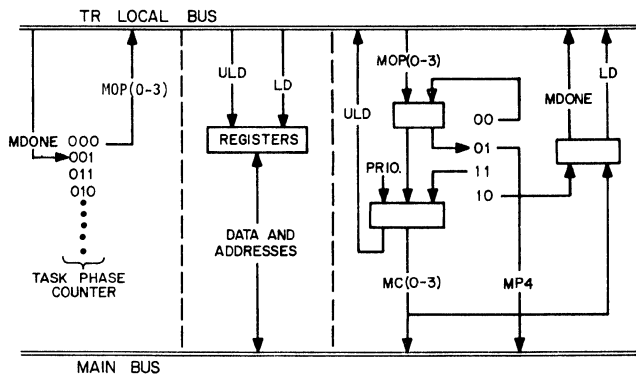


Figure 3. MC Communication Interface

### C. TR REGISTERS

The TR utilizes four Data Registers and eight Address Registers. See Figure 4. One of the basic functions of the TR is character processing (full word = 64 bits, one character = 8 bits). For this reason, the four Data Registers (Source, Name, Object, and Stack) have very flexible single character control as well as full word capability. Each Register has different capabilities depending upon the function it performs. The Source Register is used to fetch and hold the current word of the source program under compilation. It gets loaded in the full word parallel mode from the Main Data Bus, but it only outputs a single 8-bit character at a time for decoding and interpretation. The Name Register is a working register used for building and holding identifiers currently under consideration for or from the Name Table. It is the most versatile Register because it has both single character and full word capability in both its input and output. The Object Register is used to hold the Object code generated by the TR and store it in Memory. It needs only single character input capability but full word output capability. The Stack Register, used for maintaining and manipulating the stack, also has both character and word capability for input and output.

Typically, each Data Register is associated with a three-bit counter and a three-bit register to achieve character control. The three-bit register is referred to as the pointer. It gets loaded in parallel and it

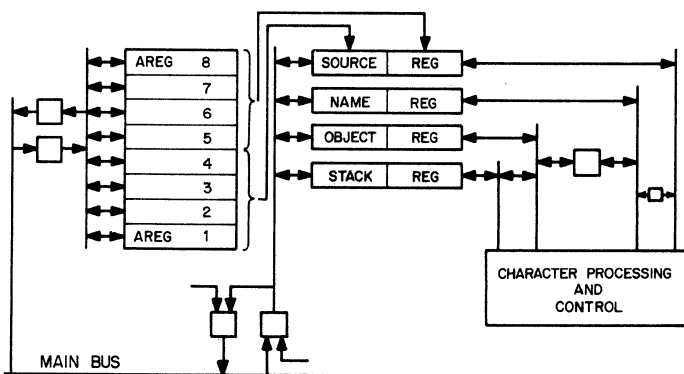


Figure 4. TR Registers

points to one of the eight characters in the Data Register for reference reasons. The three-bit counter is an up-down counter with parallel loading capability. It usually gets loaded in parallel from the pointer register. Thereafter, it responds to count-up or count-down (forward/backward) commands. The eight decoded states of the counter combined with the Read/Write command provide the selection signals for character selection in the Data Registers.

The eight Address Registers are named Address Register 1 through Address Register 8 (AREG1-AREG8). Each AREG consists of 24 bits. All eight registers communicate with the Memory. However, AREG1-AREG4 also communicate with the left half of the Data Registers (characters 2, 3, 4) and AREG5-AREG8 also communicate with the right half of the Data Register.

### IV. NAME TABLE SECTION

Most compiler systems do not use a separate name table. Address references to data space are contained in the program string.

One of the most distinguishing features of the SYMBOL compiler is the use of a separate Name Table during execution. In this way, the program string contains only references to the Name Table entry which, in turn, contains all the pertinent information and pointers, for the NAME. Any future change in the parameters will affect only the Name Table entry.

#### A. NAME TABLE CONSTRUCTION

Control is given to the Name Table logic by the Object logic section with the source register pointer pointing to the first character of the potential identifier. The Name Table logic starts searching the Reserved Word Table (RWT). If a match occurs, it puts the code on the bus and turns control back to the Object Section for processing the Code. If there is no match in the RWT, it determines the boundaries of the current word by searching and locating the next delimiter in the source string. Now, having the exact size of the identifier, it starts searching the current Name Table block. If a match occurs there, it puts the address of its Control Word on the appropriate Address Register and gives Control to the Object Section for processing. If no match occurs in the current block, the identifier is considered as local (by default) and it gets inserted at the bottom of the Name Table. Its Control Word is created in the next assigned memory location, and the address of the Control Word is placed in the appropriate Address Register. Control is now given back to the Object Section.

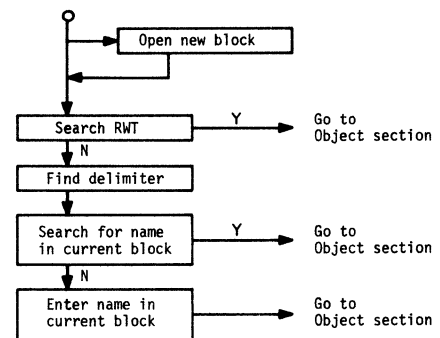


Figure 5. Name Table Construction Flow Diagram

Figure 5 shows the overall flow diagram of Name Table construction. The Name Table consists of one or more blocks that can be nested as shown in Figure 6. There is no hardware limit to the degree of nesting even though Global declarations carry identifiers up only one block level.

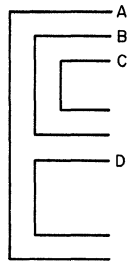


Figure 6. Block Structure of Name Table

**B. BLOCK ORGANIZATION**

The basic scheme of block organization is shown in Figure 7. There is a Block Start Control Word at the beginning of each block that contains linking and status information concerning the whole block. The body of the Block consists of VFL identifiers followed by their Control Words. The Control Word of the last identifier is properly marked to signify the end of the block. Figure 8 shows the block linking for the block structure of Figure 6. Thus, the Forward Link threads all blocks in the program, starting with the outermost block. This link is followed during the Global Linking phase in order to go through every block in the program and make sure that all identifiers are resolved as either being local to the block or global to the enclosing block or to the system (as in procedure calls, etc.). The Back Link is followed again during Global Linking to search enclosing blocks. From the outermost block there is an automatic exit to the System Name Table if there is a possibility for the identifier to be System procedure.

The basic search mechanism, from the hardware standpoint, uses two data registers and two address registers. One data register holds the name under consideration and the other holds the current name of the block being searched. A character-by-character comparison is administered until either a mismatch

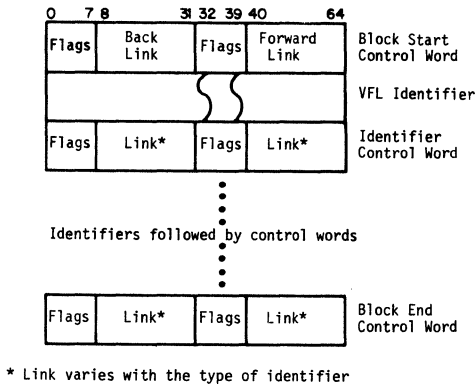


Figure 7. Name Table Block Organization

occurs or the Control Word of one identifier is reached. This means that the comparison has failed. If the Control Words of both identifiers are found simultaneously, then the comparison is successful and the appropriate linking occurs. The Address Registers are used in conjunction with a memory command (fetch and follow, follow and fetch, Store and Assign, etc.) when crossing a word boundary to fetch the next word or to store the Control Word back in memory after linking.

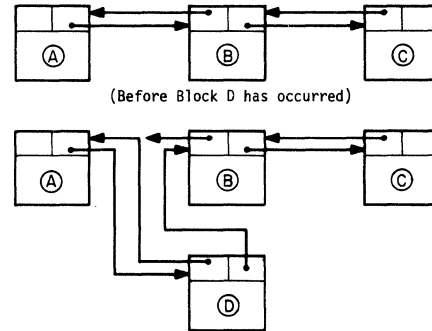


Figure 8. Block Linking Method for the Block Structure of Figure 6.

**V. RESERVED WORD TABLE**

The Reserved Word Table is a list of the words used in the internal character set as part of the SYMBOL language syntax. The table is stored in an area of the memory which is non-pageable but enjoys the automatic incrementing and link following capabilities of the MC in order to facilitate searching. The list is arranged alphabetically. Each Reserved Word occupies as many Memory words as needed.

The code for each RW is stored in the last character of the last memory word occupied by the RW. Thus, in the case of ABSOLUTE, as shown in Figure 9, the code 99 had to go in the next Memory Word. The address of the first word in the list of the RWT for each letter of the alphabet is kept in a link table that occupies the first four words of the first group of the page that holds the RWT. The table, as shown in Figure 9, is arranged so that the address of the link for each letter is directly related to the code for that letter. Thus, a portion of the code of the word's initial letter is used directly as the address to fetch the link of the first word in the RWT.

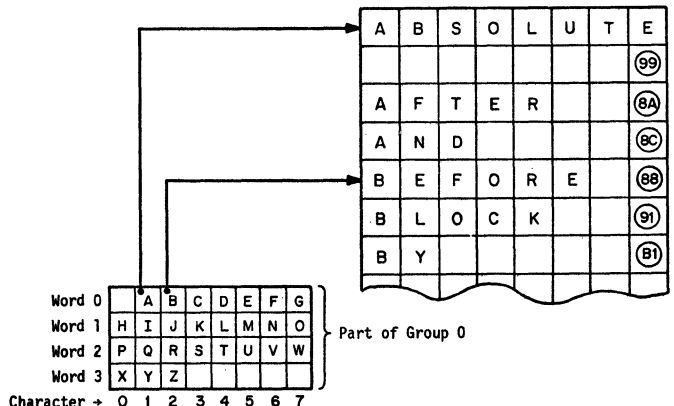


Figure 9. Reserved Word Table Organization and Linking



The code for letter A, for example, is /41 = 01000001. Thus, by using the last three bits (001) we can address directly the link for letter A which is stored at character 1 of word 0. This link will now point us to the address of the first word of the RWT that begins with A. Now we begin comparing the source program word with the RWT. If a mismatch occurs or if we reach the RW code before the end of the source word, we move on to the next RW. If the first character in the next word fails to match, then we have exceeded the list for the particular letter. Therefore, the source word under consideration is not a Reserved Word.

## VI. SYSTEM LIBRARY

The system library consists of two parts: the System Name Table, which serves as an index to the system programs, and the system programs themselves.

A system name (System Procedure) is the name of a program stored in memory as part of the system library that contains frequently used programs and service programs. There are two types of system library programs:

### A. Restricted System Programs (RSP)

A restricted system program can only be called (used) by privileged users. A privileged user is either a privileged terminal or a privileged system program.

### B. Nonrestricted System Programs (NSP)

Non restricted system programs consist of two types: Privileged System Programs (PSP) and Common System Programs (CSP).

The System Name Table consists of program names (identifiers) in the VFL form followed by one control word. The control word holds the address that points to the system program somewhere in core and also displays information about the type of system program (RSP, PSP, CSP) and the status of the Name Table at that point (Table Start, Table End).

There are two different Name Tables, one for the RSP and one for the NSP. The two main reasons for the two different tables are: flexibility of library manipulation and speed-up of search.

The address of the beginning of these tables is held in the Header area of the terminals (CH2). Thus, pos-

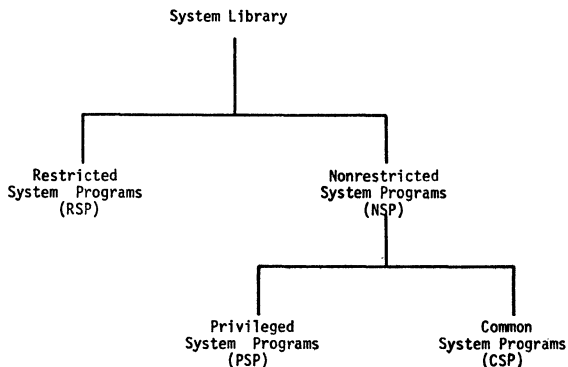


Figure 10. System Library Organization

sibly, although not necessarily, each terminal could have its own library or have no access to system library at all or a group of terminals could have the same library. This type of arrangement is primarily aimed at keeping the system library, or parts of it, out of the reach of unskilled users or users who have no need for it. It is not intended that each terminal have its own library because there will be a fair number of system programs that will be needed by many terminals. Repetition of these program names in every terminal's library will use up too much space in memory.

Referring to Figure 10 which shows the system library structure, the following observations can be made:

nonrestricted system programs (NSP) may be called by any user (terminal or a program);

restricted system programs (RSP) may be called only by privileged users;

privileged users are a privileged system terminal or a privileged system program;

an RSP or a PSP can call any other RSP or NSP;

a CSP can only call a PSP or another CSP.

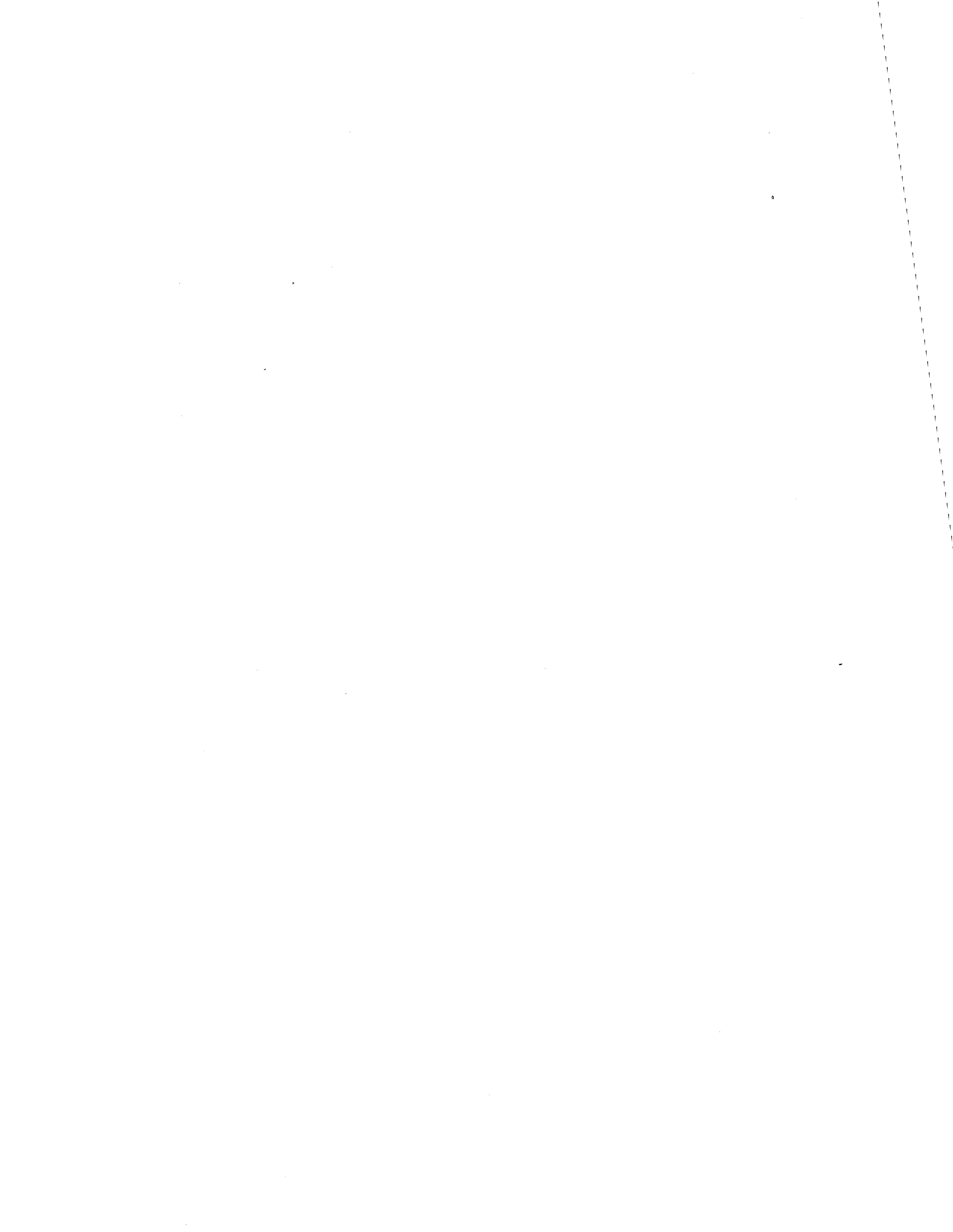
## VII. CONCLUSION

Even though a microprogram based hardware compiler would have given the system greater flexibility, the present compiler has proven that a hardware compiler was not only possible but also reasonably successful even with the technologies of the late 1960s. The software empire which grew so big so quickly in the last decade, was, for the first time, seriously assaulted by the SYMBOL compiler.

SYMBOL was meant to be an experimental machine. There are many approaches that a designer can take in implementing a hardware compiler. The SYMBOL compiler represents only one approach which in totality may or may not necessarily be the ultimate in efficiency. However, some of the algorithms developed and proven will continue to form the guidelines for some time to come.

## REFERENCES

1. SYMBOL: A Major Departure From Classic Software Dominated von Neumann Computing Systems, R. Rice, W. R. Smith, Proceedings SJCC 1971.
2. SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software, W. R. Smith, R. Rice, G. D. Chesley, T. A. Laliotis, S. F. Lundstrom, M. A. Calhoun, L. D. Gerould, T. G. Cook, Proceedings SJCC 1971.
3. The Hardware-Implemented High-Level Machine Language for SYMBOL, G. D. Chesley, W. R. Smith, Proceedings SJCC 1971.
4. The Physical Attributes and Testing Aspects of the SYMBOL System, B. E. Cowart, R. Rice, S. F. Lundstrom, Proceedings SJCC 1971.
5. System Supervision Algorithms for the SYMBOL Computer, William R. Smith, Digest COMPCON 72.



# THE ARCHITECTURE OF CASSM: A CELLULAR SYSTEM FOR NON-NUMERIC PROCESSING

George P. Copeland, Jr.  
G. J. Lipovski  
Stanley Y. W. Su  
*University of Florida*

## ABSTRACT

This paper presents the architecture of a context-addressed cellular system for non-numeric information processing, using an inexpensive, large-capacity circulating memory device. The system allows data to be represented in a structure very close to the form as the user perceives it (information structure) and allows the search operations of high level queries to be implemented directly. The information structures currently used in existing information systems are described. Then the architecture of the system as a whole is presented, as well as the implementation of these information structures as basic data types and hardware management of storage allocation and garbage collection.

The paper intends to demonstrate that distributing intelligence throughout a rotating memory device can decrease the time required for search operations in large data bases. And that the search strategy and storage management functions can be efficiently carried out in hardware, greatly simplifying the software of information systems. Thus, data not only becomes faster but easier to access, verify, insert and delete.

## 1. INTRODUCTION

Most existing information systems are implemented on general purpose von Neumann type computers. Von Neumann processors have serious inherent limitations when they are applied in non-numeric information processing. We shall discuss some of the limitations with respect to both retrieval language and storage of data in information systems.

Due to the serial nature of von Neumann processors, the time required to access a data item if nothing is known concerning its physical location varies linearly with the size of the data base. In order to minimize this access time, the information structure (structure of data as the user perceives it) is transformed into a data structure designed for efficient access on von Neumann processors. The data structure is then mapped into a machine dependent storage structure. These three levels of data representation are found to be essential in the design of a file system using von Neumann processors (Wang and Lum 1971). The intermediate access path level introduces complexities in both the storage of data and the retrieval language. Since the information structure and data structures are usually quite different, the storage structure does not closely resemble the format of data as the user perceives it. Also, complicated procedural steps are introduced which are basic operations of von Neumann processors rather than basic operations of the high

level language of the user. These additional structures and procedural steps are both greatly complicated by the need to schedule paging of large amounts of information between discs (where the data base must be stored) and the primary memory of the processor. It is now widely accepted that provisions for isolating the user from the data and storage structure levels is one of the major objectives of a data base system (CODASYL report 1971a-b, Engles 1970, and Guide/Share report 1971). However, this has proven to be a very difficult and expensive task in software. This paper presents a hardware solution to these basic problems. The following paragraphs indicate the approach we have taken.

If all operations on the data base are done directly in (fixed head) disc memory where the entire data base is stored, then the excessive paging is eliminated. Also, parallelism is used to make the time to search the data base independent of the data base size. This eliminates the need for an intermediate access path level, because the entire data base is searched by hardware for each search instruction. Thus, the parallelism inherent in high level retrieval languages can be implemented without the need to translate the specification of what is desired by the user into complicated procedural steps. Data can be stored in a format which is very close to the user's information structure, removing the data representation at the access path level from the task of data definition by the user.

The idea of using distributed intelligence in inexpensive, large capacity, circulating memory devices has evolved slowly. Partially associative devices have been suggested (Hollander 1956, Parker 1971, Minsky 1972). They allow name-value pairs as the basic data type and allow only the name part to be searched. Content associative devices (Fuller et.al. 1965, Parhami 1972) allow the value part to be searched also. String, substring and template searches have been examined (Healy et.al. 1972) on a context addressed disc. The context-addressed, segment-sequential memory (CASSM) described here offers several advantages over these devices. It allows widely used information structures (such as trees, sets, graphs and relational tables) to be implemented as basic data types of the machine. Also, the task of storage allocation and garbage collection is taken over by hardware. A more detailed discussion of software advantages and considerations is given in Su et.al. (1973)

In section 2 we describe the various information structures widely used in non-numeric processing. In section 3, the hardware of the CASSM which implements these high level data types and automatic storage allocation and garbage collection is presented. A

summary and conclusions are given at the end.

2. INFORMATION STRUCTURES OF NON-NUMERIC PROCESSING

In non-numeric processing, several information structures have become useful in representing information. They are the directed graph or network model (CODASYL 1971a), the relational model (Codd 1970, 1971) and the tree or hierarchical structure, which is commonly used in data processing systems. Information is represented in each of these structures in general as a set (record) of attribute-value pairs in each node or table entry. These are called information structures because the user views his data as being displayed most naturally in these structures, and because operations of his data involve specifying parameters that are also parameters of the structures.

In order to see more concretely which operations are performed on these information structures, let us examine an example inventory file taken from J.C. Date (1972) in his tutorial description of Codd's work on relational files. The logical relations among the data fields in the file can be represented by a tree structure as well as by a network structure. They can also be represented by E.F. Codd's (1970) normalized relational form involving three relational tables as in Figure 1. The tables show a many-to-many mapping of suppliers and parts (each supplier supplies many parts and each part is supplied by many suppliers). Each table defines a relation with the domains of the relation shown as the headings of the columns. Each row contains a set of attribute (name)-value pairs, where the attributes are listed as domain headings. In the example, Date shows four possible queries to be satisfied:

- (a) find part numbers for parts supplied by supplier 2;
- (b) find part names for parts supplied by supplier 2;
- (c) find supplier numbers and status for suppliers in London;
- (d) for each part find part number and names of all cities from which the part may be obtained.

In Figure 1, the supplier-part (SP) table shows how the many-to-many mapping is handled in the relational model using redundant data values to link tables by contents rather than by addresses. If the user is supplied with the skeletal description of the table arrangement as in Figure 2, he may specify each of the above queries in a non-procedural, calculus type statement similar to that developed by Codd (1971) and given in Date:

- (a)  $SP.P\# : SP.S\# = 2$
- (b)  $P.PNAME : \exists SP((P.P\# = SP.P\#) \wedge (SP.S\# = 2))$
- (c)  $S.S\#, S.STATUS : S.CITY = 'LONDON'$
- (d)  $P.P\#, S.CITY : \exists SP((P.P\# = SP.P\#) \wedge (S.S\# = SP.S\#)), \forall P.P\#.$

The data items in the above statements are specified by qualified names as those used in COBOL and PL/1. The expression on the left of the colon specifies what is to be retrieved and the expression on the right is a qualification. For example, the first statement is a query for retrieving all part numbers (P#) supplied by supplier number 2 (S#=2).

This same inventory file can be put into a hierarchical form in which suppliers are superior to parts as in Figure 3. If the user is given the skeletal description of the file as in Figure 4, he may express each of the queries in a non-procedural, calculus type statement as follows:

- (a)  $S.P.P\# : S.S\# = 2$
- (b)  $S.P.PNAME : S.S\# = 2$
- (c)  $S.(S\#, STATUS) : S.CITY = 'LONDON'$
- (d)  $S.(P.P\#, CITY) \forall S.P.P\#$

The qualification on the right of the colon is simplified by the fact that specific hierarchical dependencies are given in the specification on the left using qualified names and parentheses.

Statements like the queries above can be either implemented as basic operations of CASSM or easily broken down into basic operations. The details of how this is done is included in section 3.2 and 3.3.

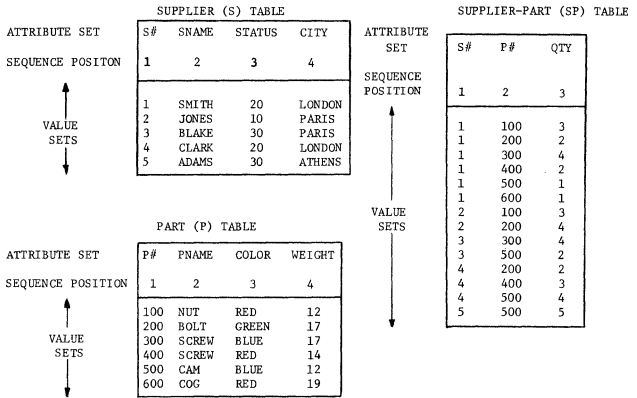


FIGURE 1. INVENTORY FILE IN CODD'S NORMALIZED RELATIONAL FORM

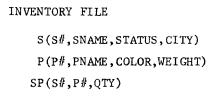


FIGURE 2. SKELETAL DESCRIPTION OF FILE IN CODD'S NORMALIZED RELATIONAL FORM

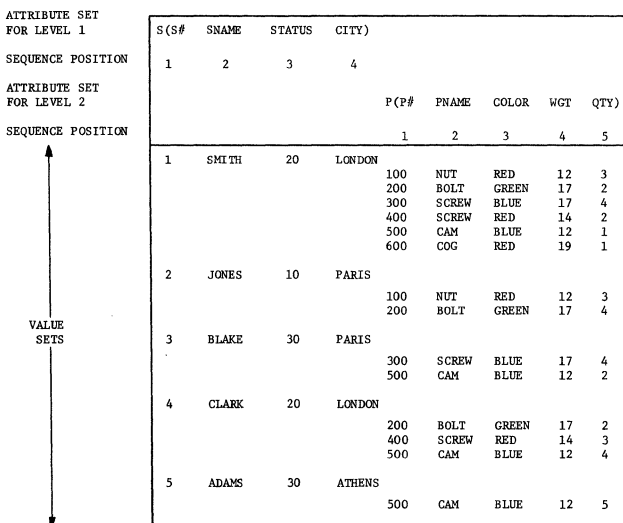


FIGURE 3. INVENTORY FILE IN A HIERARCHICAL FORM

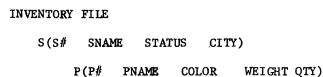


FIGURE 4. SKELETAL DESCRIPTION OF FILE IN A HIERARCHICAL FORM

### 3. GENERAL DESCRIPTION OF HARDWARE

In order to fully exploit LSI technology, CASSM consists of a chain of identical cells. Each cell can communicate directly with its two neighboring cells and with an IO bus common to all cells. Each cell consists of two parts: a circular, sequential segment of memory (such as, a disc track, a circular charge-coupled device, or a magnetic bubble device) and a logic section. All segments of memory circulate concurrently and in synchronization, while each logic section reads, searches, modifies and rewrites its segment of memory from one end to the other. Thus, all segments of memory are operated on in one circulation of memory. A read and a write head per track (segment) is required for implementation on a set of discs. The conceptual arrangement of the hardware in each cell is illustrated in Figure 5. The remaining sections of this paper describe the function and implementation of the submodules of this figure.

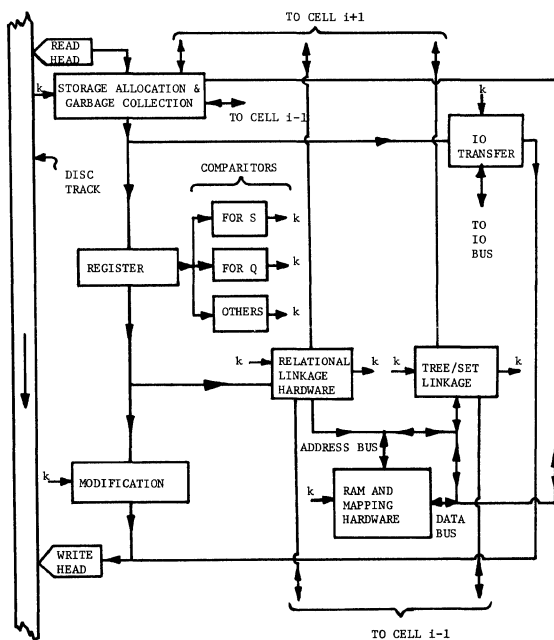


FIGURE 5. CONCEPTUAL HARDWARE LAYOUT OF CELL 1  
(k's INDICATE CONTROL LINES WITHIN THE CELL)

The regularity involved in having identical cell logic and all memory segments equal in length is desirable for cost effective hardware implementation. However, information structures used in non-numeric processing are highly variable in length. To require the user to partition his structures to fit into these equal length memory segments would lead to inefficient utilization of memory and to greatly increased software costs. In CASSM, variable length structures are divided into equal length segments for high utilization of memory as in Figure 6. Each segment may contain only a part of a record, a whole record, or several records of a file. Submodules within each cell of Figure 5 allow operations on variable length structures that overlap any number of segments. The forward and backward marking facility provided by a one bit random access memory (RAM) will be described in the following section.

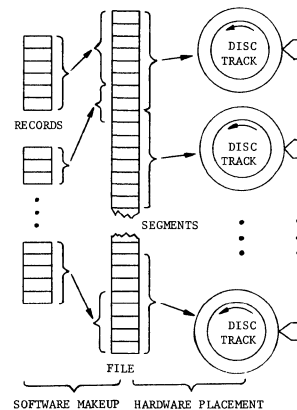


FIGURE 6. STORAGE OF RECORDS AS SEGMENTS

#### 3.1 Forward and Backward Marking

In section 2, we pointed out that high level (non-procedural) statements have two distinct parts: a specification (S) of what is to be marked and a qualification (Q) that must be met for the marking to take place. Each query involves searching and conditionally marking all occurrences of S-Q pairs throughout the data base. If the occurrences of the pairs overlap one another, i.e., if any element of a pair occurs in between the elements of another pair, then the search would take more than one disc revolution. If the data base is stored such that for each query, each occurrence of an S-Q pair referred to by the query does not overlap any other pair in the sequence, then the pairs can be operated on one at a time as memory sweeps by. We need only enough hardware to operate on one pair at a time. Using two comparators within each cell (one for S and one for Q, as in Figure 5) allows S and Q to be searched for during the same sweep of memory. However, these two parts may be separated by an arbitrarily long distance with much data in between. If Q occurs before S in the sequence (forward marking), this does not present any problem to implement. The one information bit regarding the success of the Q search can be saved until S is found and conditionally marked later in the sequence. But if S occurs first, then it cannot be marked until Q is found and satisfied later in the sequence. We need some way to access a mark bit of S after Q is searched (backward marking).

This can be accomplished by having a set of mark bits that can be accessed independently of the position of the circular memory segment and with a simple hardware method of mapping the mark bits to data items on the segment. Figure 7 shows how a small one bit wide RAM within each cell is used to do this. A counter initially set to zero at the beginning of each segment revolution is used as a hardware pointer to the one bit RAM. The beginning of each data item indicates that the counter is to be incremented (using a special delimiter bit or symbol) so that the counter points to a unique marker bit for each data item in a segment. We have a 1-1, onto mapping of marker bits to data items. Although a RAM is being used, data items are not tied down to a physical location and items may be of variable length. Only their relative positions in the sequence are important.

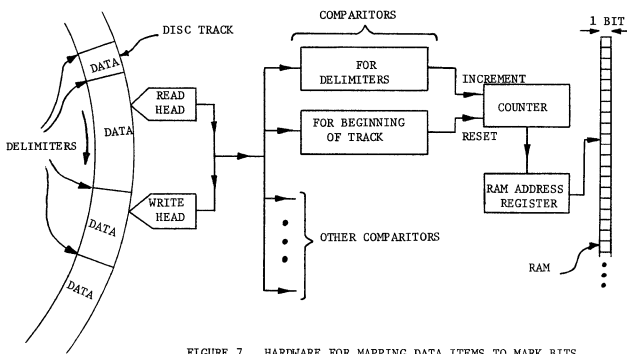


FIGURE 7. HARDWARE FOR MAPPING DATA ITEMS TO MARK BITS

### 3.2 Implementation of Information Structures

In this section we shall describe the implementation of the various information structures widely used. The organization of data and the search operations in the disc system will be described.

#### 3.2.1 Trees or Hierarchical Structures and Sets

Information is represented in a tree or hierarchical structure as a set, record or tuple of attribute (name)-value pairs in each node of the tree. A set is linearized by simply listing each set member sequentially. A set member can then be accessed by its attribute or position in the sequence and by its value. To greatly improve storage efficiency, sets are of two types, attribute sets and value sets. An attribute set can be placed in front of each value set or in front of a large number of value sets which have the same set of attributes. Thus storage efficiency is improved by not repeating identical attribute sets. A sequence position counter CSP, incremented by data item delimiters as in Figure 7 but reset by beginning set delimiters, indicates which set sequence position is currently being examined. If set members are accessed by their attribute, then two segment revolutions are needed. During revolution 1, the specified attribute is searched for in the attribute sets and marked whenever found. During revolution 2, the sequence position of the marked attribute (provided by CSP) is saved in a register RSP and compared to CSP during examination of the subsequent value sets. Value sets need not be stored in the same cell as their attribute set. Some value sets may be separated by several cells from their attributes. The following procedure allows hardware communication of the sequence position number from one cell to the following cells.

During revolution 1 above, whenever an attribute is found to match, the sequence position in CSP is stored in RSP. Between revolutions, the contents of RSP in a cell containing an attribute match is propagated to the following cells. A one bit register R1 is used to cut off the propagation of the sequence position at the cell which contains the last item in the subtree that is presently being searched. R1=1 indicates that the cell contains either a value set which has a level number less than the level number presently being searched or an attribute set which has a level number equal to the one presently being searched. Between revolutions, the contents of RSP in each cell is sent to cell  $i+1$  and stored in the RSP of cell  $i+1$ . If R1=0 in cell  $i+1$ , (indicating that the cell is not at the end of the subtree) then the contents of RSP of cell  $i+1$  is also sent to RSP of cell  $i+2$ . The procedure is repeated until the sequence position of the specified attribute reaches the end of

its range over segments. Thus the correct sequence position number is prestored in the RSP of each cell before revolution 2. Revolution 2 is then executed exactly as described above. One bit of communication between adjacent cells is required.

A tree can be linearized in several ways (Knuth Vol. 1, 1969). If the tree is written in preorder with level numbers included with each node, then it is uniquely specified. The tree level number becomes part of the addressing specification. Also, for a given node at level  $\ell$ , its entire subtree is listed before the next occurrence of another node at level  $\ell$ . Thus a node at level  $k$  and any member of its subtree at level  $\ell$  ( $k < \ell$ ) form an S-Q pair that does not overlap with any other pair at levels  $k$  and  $\ell$ . This provides a convenient method for marking forward (down the tree) or backward (up the tree). For example, all ancestor nodes at level  $k$  can be marked if a search within one of its successors at level  $\ell$  ( $k < \ell$ ) is successful. This can be done using the backward marking facility of the one bit RAM in the following way. We consider first the operation within one segment of memory.

The ancestor is encountered first because of the preorder in which the tree is stored. The RAM address of the ancestor mark bit is saved for reference until the successor is searched later in the sequence. If the search is successful, then the ancestor mark bit is set using RAM address that was saved. With the tree stored in preorder together with level numbers, the above algorithm simply involves remembering the RAM address of the last node at level  $k$ . Marking forward (down the tree) is much simpler. If a descendant is to be marked whenever an ancestor is successfully searched, the only thing to be remembered is whether or not the search was successful. The same hardware communication can be used between set members within the same tree node since any two members of the same set from an S-Q pair which does not overlap any other such pair in another set. Three one bit registers (RQ, R and RS) and two 10 to 12 bit registers (RB and RF) compose the basic hardware needed in the tree/set submodule of Figure 5. The functions of these registers are described below.

For forward marking RQ is used to save the information regarding the success of the Q search. For backward marking, RB is used to save the RAM address of the most recently encountered occurrence of an S. For both forward and backward marking, logic is needed to load either the counter of Figure 7 (forward) or RB (backward) into the RAM address register, set the bit, and initialize the registers for the next pair.

We now consider the operation on multiple segments. Although the above hardware is sufficient for processing sequentially encountered S-Q pairs residing on the same track, the elements of some pairs may be located on different tracks. In order to process these pairs in one revolution of memory, additional hardware is necessary.

The following is needed for forward marking across tracks. RS is used to indicate at the end of a circulation of memory whether a Q (first pair member) has been satisfied on a segment without encountering an S after it on the same track. Thus RS indicates that the S occurs on one of the following tracks. RF is used to save the RAM address of the S if found in a following track for marking at the end of the circulation of memory. This is necessary because we cannot be sure that the Q on the previous segments has been satisfied until all memory has been searched. R is used to indicate whether at least one occurrence of S has been found in a segment. Also, one bit of communication is required between adjacent cells. The hardware procedure

is as follows. If RS=1 in cell  $i$  at the end of a circulation of memory, a pulse is sent to cell  $i+1$ . If R=1 in cell  $i+1$ , RF in that cell is used to set the mark bit of S. If R=0, the pulse is sent on to cell  $i+2$ . This is repeated until the cell is reached that has R=1. The marking is then done using the RF of that cell.

Much of the same hardware may be used for backward marking across tracks, since only one mode (forward or backward) is involved in each instruction. RS is used to indicate whether the elements of an S-Q pair reside on different tracks. R is used to indicate whether at least one occurrence of Q has been found in a segment. RB will be used to save the RAM address of S. Since this is the last S encountered, its address is always present in RB at the end of a circulation of memory. The same communication bit between cells is used except the pulse is sent in the opposite direction. With the exception of these changes, the hardware procedure is the same for forward and backward marking.

Thus we have the capability of marking a node or node member if another node or node member satisfies a given condition. This can be done in one disc revolution if the communicating elements are not in different subtrees.

### 3.2.2 Tables, Graphs and the Relational Data Structure

The tree/set hardware can also be used to aid in implementing tables like those of Figure 1 by providing a means of communicating between elements within each table. If a table is at tree level  $i$ , then each row (set) in the table is at level  $i+1$ . All data items in the same row are members of the same tree node set. Also, several tables may be grouped hierarchically.

General graphs or networks cannot be linearized. However, they can be implemented using tables in two ways. One way is to set up a table for each node of the graph (16). Each table would contain as rows the node names of nodes pointed to by the table node along with their corresponding relation or arc names. Alternatively, a table can be set up for each relation or arc name. Here each table would contain as rows the node name pairs that are connected by the table relation or arc. Relations of degree  $n$  can be stored by allowing more than two columns, using the set hardware. Figure 1 is an example of a set of tables, based on relations rather than node names, where each table corresponds to a relation name and each row is a set of nodes that are related by the table relation. In this information structure, cross references between tables are specified by content rather than by using physical address pointers. Query (b) involves communication between tables. The execution steps are as follows. First, the command SP.P# : SP.S# = 2 is executed using the tree/set hardware. Secondly, the marked SP.P#'s are used to mark rows in table P having the same P#. This requires the communication between tables implied by P.P# = SP.P#. Finally, the PNAME's within these rows are marked. This is accomplished by the tree/set command P.PNAME : P.P#. Two methods are described below that accomplish the communication between tables in the second step.

In both of the methods described below, there exists the need to distinguish the marked source data items from the newly marked destination data items. Otherwise, the destination items may be used as source items. This can be done by having two sets of mark bits, one for source items and the other for destination items. Only the set of mark bits for destination items need be independent of position of the disc. The mark bits for source items can be stored

on the disc along with the items. A method would be needed in hardware to allow these two sets of mark bits to conditionally set one another. Also, in both of the schemes described below, a method is needed to indicate when all data items have been traversed. This is done by resetting the mark bit of each source item when it is used, and employing an OR rail (12 and 16) between cells to indicate whether any mark bits are still set. The most obvious method for traversal between tables is to pick up the marked node names to be traversed from the source table and use these names to context search the destination tables. This uses only the tree/set hardware. However, if many items are to be traversed, either many comparitors are needed or many revolutions (searches) of memory are necessary. The only additional hardware needed to implement this method is in the form of additional comparitors to speed the searching if desired.

The second method is to prestore in each potential source table the RAM addresses of the mark bits of node names in the destination table. In Figure 1, columns with attribute S# in tables S and SP are cross references between tables. Using this method, the RAM addresses of the SP.S#'s are stored next to the corresponding S.S#'s. Similarly, pointers are stored in tables P and SP to cross reference items under attribute P#. These pointers can be prestored by picking up each node name in one table and using it to context search the other table as in the first method. Except here, each time the search is completed, the pointers stored. During the traversal in query (b), the marked pointers stored next to the SP.P#'s are picked sequentially and used to mark the P.P#'s. The additional hardware needed to implement this scheme is described below.

No additional hardware would be needed if the stored source pointers were within the same segment of memory as the data items they referred to. As each pointer is accessed sequentially, it would simply be loaded into the RAM address register. However, this is usually not the case since sizable structures will be segmented over many cells. This scheme can be extended to handle the general case, where the source pointers lie in different segments, by viewing all data items in the data base as a sequence of items. For example, cell 1 might contain items with global sequence addresses 0 to 811, items 812 to 1512 in cell 2, and items 1513 to 2301 in cell 3, etc. This concept is implemented in hardware by using a register RBSA in each cell to store the beginning global sequence address of the cell, a register RGSA to store the global sequence address, and an adder to compute RGSA by adding RBSA and the counter of Figure 7. The last sequence address of a segment is provided at the end of each revolution by RGSA and stored in register RLSA. As each pointer is accessed, its value is compared to both RBSA and RLSA. If the pointer is within these bounds, the RAM address register is loaded with the pointer minus (using the above adder) RBSA and the RAM mark bit is set. If the pointer is greater than RLSA, it is sent to the following cell for comparison. If the pointer is less than RBSA, it is sent to the previous cell for comparison. This procedure is continued until the pointer has migrated to its destination cell. A register RP is used to hold the pointer and a one-bit register indicates whether RP is occupied. Two comparitors and two additional bits of communication between adjacent cells (one bit for each direction of pointer migration) are needed. If the register holding the pointer is occupied, a newly encountered marked pointer on the memory segment may have to be passed over until the next revolution. Thus this scheme may take more than one revolution to do all the marking required. However, the number will be much smaller than that of the first

method if many items must be traversed between tables.

### 3.3 Execution of High Level Queries

This section provides a more detailed illustration of the hardware steps required to execute high level queries. The storage structure to be queried is presented first.

The relational information structure of Figure 1 is stored in a very stright-forward way as shown in Figure 8, where each row is a set. Sets may contain a variable number of information items. The level numbers show the hierarchical dependency of the rows to their table name. The number in parentheses to the left of each data item is not actually stored but is inserted in the figure to indicate the global sequence address of the item. The numbers in parentheses to the right of each data item are a list of the global sequence addresses (pointers) to which the item is linked. Further details of the storage structure, such as whether items should be stored as variable length character strings or as fixed length code numbers, will not be discussed in this paper.

SET TYPE	LEVEL NO.	INFORMATION FIELDS				
A	1	(1)S				
A	2	(2)S#	(3)SNAME	(4)STATUS	(5)CITY	
V	2	(6)1 (30,33,36,39,42,45)	(7)SMITH	(8)20	(9)LONDON	
V	2	(10)2 (48,52)	(11)JONES	(12)10	(13)PARIS	
V	2	(14)3 (54,57)	(15)BLAKE	(16)30	(17)PARIS	
V	2	(18)4 (60,63,66)	(19)CLARK	(20)20	(21)LONDON	
V	2	(22)5 (69)	(23)ADAMS	(24)30	(25)ATHENS	
A	1	(26)SP				
A	2	(27)S#	(28)P#	(29)QTY		
V	2	(30)1 (6)	(31)100(77)	(32)3		
V	2	(33)1 (6)	(34)200(81)	(35)2		
V	2	(36)1 (6)	(37)300(85)	(38)4		
V	2	(39)1 (6)	(40)400(89)	(41)2		
V	2	(42)1 (6)	(43)500(93)	(44)1		
V	2	(45)1 (6)	(46)600(97)	(47)1		
V	2	(48)2 (10)	(49)100(77)	(50)3		
V	2	(51)2 (10)	(52)200(81)	(53)4		
V	2	(54)3 (14)	(55)300(85)	(56)4		
V	2	(57)3 (14)	(58)500(93)	(59)2		
V	2	(60)4 (18)	(61)200(81)	(62)2		
V	2	(63)4 (18)	(64)400(89)	(65)3		
V	2	(66)4 (18)	(67)500(93)	(68)4		
V	2	(69)5 (22)	(70)500(93)	(71)5		
A	1	(72)P				
A	2	(73)P#	(74)PNAME	(75)COLOR	(76)WEIGHT	
V	2	(77)100(31,49)	(78)NUT	(79)RED	(80)12	
V	2	(81)200(34,52,61)	(82)BOLT	(83)GREEN	(84)17	
V	2	(85)300(85,55)	(86)SCREW	(87)BLUE	(88)17	
V	2	(89)400(40,64)	(90)SCREW	(91)RED	(92)14	
V	2	(93)500(43,58,67,70)	(94)CAM	(95)BLUE	(96)12	
V	2	(97)600(46)	(98)COG	(99)RED	(100)19	

FIGURE 8. STORAGE STRUCTURE OF CODD'S RELATIONAL TABLES (EACH ROW IS A SET)

The comparator submodules for S and Q in Figure 5 are identical. A simplified illustration of the hardware of one of these submodules is given in Figure 9, as an example of how the hardware might be arranged. A more detailed discussion of the hardware arrangement of the comparator submodules will not be given in this paper. The three comparators are one-bit, serial adders, capable of arithmetic inequalities as well as exact matches. Each comparator sets a flip flop when the specified comparison is successful. The FF's are reset before each item is searched. A MATCH is the logical AND of these three FF's. A one-bit register ROR is used to allow an ordered search of items. ROR is set when a marked item is encountered and reset if a tree level number is encountered which is both not marked and less than the level number being searched. Thus, ROR allows the ordered set search to remain with the previously specified subtree. This ordered set search is allowed to work over one or more segment boundaries as described by Healey (9) except that one item is searched per revolution instead of one character.

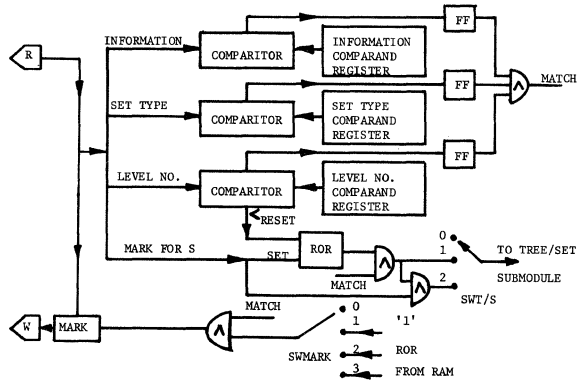


FIGURE 9. SIMPLIFIED HARDWARE ARRANGEMENT OF THE S COMPARITOR SUBMODULE

A simplified microcode is given for S and Q in Figure 10 to execute query b. The contents of the three comparand registers are given, as well as the position of the two switches in Figure 9, for each revolution of memory. Mark bits which were set during previous revolutions are reset if their data items do not satisfy the new marking conditions. Revolutions 1, 2 and 3 execute SP.P# : SP.S# = 2 (same as query a). Revolution 4 transfers the mark bits of the RAM to the storage dependent mark bits for Q. Revolution 5 executes the marking between tables indicated by P.P# = SP.P#. Revolution 6 transfers the mark bits of the RAM to the storage dependent mark bits for Q. Revolutions 7 and 8 execute PNAME : P#.

Thus, CASSM executes a rather complex example query in 8 segment revolutions, or approximately 80 ms for a disc. Furthermore, this time is independent of the data base size. Non-numeric information systems implemented on von Neumann computers must page bulk information (much of which is not relevant to the query) from discs to primary memory, requiring much time and expensive channels.

SEG REV	S					Q				
	SET TYPE	LEVEL NO.	INFOR.	SWMARK	SWT/S	SET TYPE	LEVEL NO.	INFOR.	SWMARK	SWT/S
1	A	1	SP	1	0	A	1	SP	1	0
2	A	2	P#	2	0	A	2	S#	2	0
3	V	2	(DON'T CARE)	0	1	V	2	2	0	1
4				0	0	(DON'T CARE)	(DON'T CARE)	(DON'T CARE)	3	0
5	MARK	RAM	BITS FROM POINTERS OF ITEMS USING THE RELATIONAL LINKAGE			(DON'T CARE)	(DON'T CARE)	(DON'T CARE)	3	0
6				0	0	(DON'T CARE)	(DON'T CARE)	(DON'T CARE)		
7	A	2	PNAME	1	0	A	2	P#	1	0
8	V	2	(DON'T CARE)	0	1	V	2	(DON'T CARE)	0	2

FIGURE 10. SIMPLIFIED MICROCODE FOR QUERY b: P.PNAME : SP((P.P# = SP.P#) ^ (SP.S# = 2))

### 3.4 Storage Allocation & Garbage Collection (SA & GC)

From the software point of view, we would like to be able to initially load the data base and insert and delete items without regard to physical location. We would like to free the programmer from the burden of accounting for what data is in which segment of memory or its position within that segment. The only aspect of SA and GC that the programmer need be aware of should be a warning that the data base has exceeded the total size of memory. Insertions and deletions should be no more complicated than specifying where in the user's



information structure to insert or delete and the information to be inserted. In CASSM, associative garbage instructions can be used to mark where to insert or delete based on context within the information structure rather than physical location. The task of making room for new data and repacking memory when holes are left by deletions can be done automatically in hardware. The scheme to do this is described below.

Two registers RVL and RT are the basic hardware of the SA and GC submodule in Figure 5. As the read head picks up data, it is fed into one end of a shift register RVL which shifts at the same bit rate as the memory segment. A tap is provided for the output of RVL at multiples of W from the input, where W is the basic word size of the machine. When storage is not being allocated or collected (Figure 11-b), the write head uses the center tap of RVL as its input. When garbage is being collected (Figure 11-a), the input of the write head moves over one tap toward the input of RVL each time a word marked for GC is encountered on a segment. This eliminates that word from the sequence in memory. If RVL is not long enough to collect all words marked for GC on that segment, they can be collected in subsequent revolutions. When storage is being allocated (Figure 11-c), the input to the write head moves away from the input of RVL, again one tap for each word inserted. If RVL is not long enough to allocate all words required by an insertion, then the last word inserted can be marked so that the remaining words can be inserted beginning at that point during subsequent revolutions. The contents of the one bit RAM can be shifted forward or backward from the point of insertion once for each RAM delimiter within the insertion or deletion. This scheme provides SA and GC within each cell. However, the number of words used within a cell may grow too large for a segment to hold, or so small that much memory is wasted. A method of managing data transfers between cells is described below.

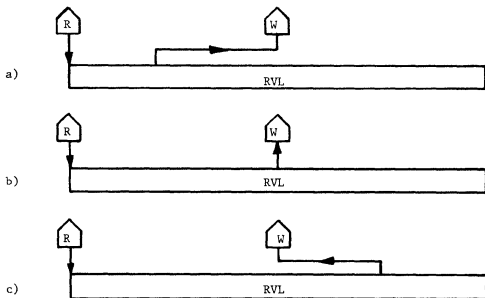


FIGURE 11. VARIABLE LENGTH SHIFT REGISTER FOR INSERTION AND DELETION

We choose to pack data toward one end of the chain, leaving unused cells at the other end. In order to reduce the time required for providing space for insertions, some of the available memory is distributed among the cells to act as a buffer. Within each cell, data is packed toward the beginning of the track. A special tag E is used to indicate the end of the used portion of the segment. A register RT is used to act as a buffer storage for transfers between cells. When the number of words used in a cell is too large, RT is filled with the last words before the tag E. These are then stored at the beginning of the next revolution. When the number of words used in a cell is too small, RT is filled with the beginning words of the following track. They are then stored directly in front of the tag E. The register RVL is used in both cases to move the bulk of the used data forward or backward within a

cell. Also, the contents of the one bit RAM can be shifted with the data. The counter of Figure 7, which counts the number of RAM delimiters in a cell, will point to the last bit used in the RAM at the end of each revolution. The number of delimiters being transferred in RT indicates how many RAM bits to transfer, starting where the counter points. RBSA is updated by incrementing RBSA once for each delimiter passed into a cell, to or from other cells. It is possible for the size of RT to be too small to handle the number of insertions required in one revolution. In this case, the remaining insertions must be made during subsequent revolutions as the SA and GC hardware allows. Large insertions and deletions can be made in the middle of the data base by writing entire tracks forward or backward, one track per revolution.

If the data stored on the segments is free of parameters regarding physical location, the above scheme can be carried on during the same revolutions (in a pipeline fashion) as instruction execution. Also, no software intervention is needed for SA and GC. The tree/set scheme presented in section 3.2.1 has this property as well as the first method presented in section 3.2.2 for implementing communication between tables. The second method in section 3.2.2 requires that pointers, which are dependent only on the global sequence address of the data they point to, be stored in the segment memory. They require maintenance after insertions or deletions are made. However, movement of data between cells can be carried on in a pipeline with instruction execution since pointers are independent of cell number.

If an insertion is made between items with global sequence address a and a+1, then all pointers referring to items above a+1 must be changed. The pointers P are altered in parallel as indicated by the following conditional assignment:

$$P \leftarrow P + N_D \quad \text{IF } a < P, \forall P,$$

where  $N_D$  is the number of delimiters or data items within the insertion. For deletions,  $N_D$  is subtracted. If insertions are made at two points in memory, one after address a and one after address b, the following two operations are necessary:

$$P \leftarrow P + N_{Da} \quad \text{IF } a < P, \forall P;$$

$$P \leftarrow P + N_{Db} \quad \text{IF } b + N_{Da} < P, \forall P,$$

where  $N_{Da}$  and  $N_{Db}$  are the number of delimiters within insertions a and b. For n points of insertion, n such steps are necessary. Instructions implementing the above algorithm can be provided to allow this updating after insertions and deletions to be done in a simple way, and to allow full parallelism to be exploited.

#### 4. SUMMARY AND CONCLUSIONS

This paper presents the architecture of a context-addressed, segment-sequential memory designed for non-numeric information processing. Since it is unrealistic to describe the design or evaluation of a hardware system out of context of software and application, the information structures and retrieval operations currently used in the existing information systems are first described. The architecture of the system is then described to show how various information structures can be represented and search operations can be carried out directly on bulk memory with little intervention from the central processor. Hardware storage allocation and garbage collection techniques used in the system are also detailed.

The CASSM processor provides data processing capabilities useful for information retrieval in large data bases. It offers a much more cost-effective non-numeric processing system than conventional information systems which use von Neumann processors to perform search, store, arrangement, allocation, garbage collection and other data processing functions. The characteristics and advantages of CASSM can be summarized as follows:

(1) In non-numeric processing, it is extremely time-consuming to page data in and out of the secondary memory and to perform searches by the central processor, especially when the data base is large. CASSM allows data to be searched in parallel on a set of circulating devices, so that search time is independent of the size of the data base. This operation can be carried out independently of the central processing unit.

(2) In non-numeric processing, the user shall be allowed to work with the data as he sees it (i.e., at the information structure level) without having to concern himself with the internal representation of the data. CASSM allows information structures to be stored as they are without going through many levels of data mapping which are found necessary in conventional computers to achieve search efficiency. High level search queries specified by the information user can be performed by the memory device as basic operations, thus simplifying the retrieval language design. This feature of CASSM avoids many problems found in information systems concerning data reliability, excessive storage requirement and structure construction and maintenance.

(3) The data base of an information system is generally dynamic in the sense that the contexts are constantly changing. Considerable amount of insertions, deletions and modifications need to be performed. Memory management is a serious problem and is generally handled by software using CPU time. In CASSM, management of memory is greatly simplified by having only one level of memory hierarchy and by the SA and GC hardware.

(4) From the hardware point of view, CASSM offers several advantages. The class of sequential memories can be very inexpensive. Also as a data base grows, more memory units can be added modularly. The logic requires only one LSI chip type and interconnections are very simple and regular because all the cells are identical.

#### REFERENCES

1. CODASYL Systems Committee: "Introduction to 'Feature Analysis of Generalized Data Base Management Systems'", CACM 14,5 (May 1971b), pp. 308-318.
2. CODD, E.F., "A Relational Model of Data for Large Shared Data Banks," CACM 13,6 (June 1970), pp. 377-387.
3. CODD, E.F., "A Data Base Sublanguage Founded on the Relational Calculus," Proceedings of ACM SIGFIDET Workshop on Data Description, Access and Control, (Nov. 1971), pp. 35-68.
4. Data Base Task Group of CODASYL Programming Languages Committee: Report, April, 1971a.
5. DATE, C.J., "Relational Database Systems: a Tutorial," paper presented at the Fourth International Symposium on Computer and Information Sciences, December 1972.
6. ENGLER, R.W., "A Tutorial on Data Base Organization," IBM Technical Report TR 00.2004, IBM, Poughkeepsie, New York, March, 1970.
7. FULLER, R.H., BIRD, R.M. and WORTHY, R.M., "Study of Associative Processing Techniques," AD-621516, August 1965.
8. Guide/Share Data Base Task Force, "Data Base Management System Requirements," Share, Suite 750, 25 Broadway, N.Y., November 1971.
9. HEALY, L.D., DOTY, K.L., and LIPOVSKI, G.J., "The Architecture of a Context Addressed Segment Sequential Storage," Proceedings of FJCC, Vol. 41, part I, 1972, pp. 691-702.
10. HOLLANDER, G.L., "Quasi-Random Access Memory Systems," Proceedings of EJCC, 1956, pp. 128-135.
11. KNUTH, D.E., Fundamental Algorithms, Vol. 1, Addison-Wesley, 1969.
12. LEE, C.Y. and PAUL, M.C., "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," Proc. IEEE, Vol. 51, 1963, pp. 924-932.
13. MINSKY, N., "Rotating Storage Devices as Partially Associative Memories," Proceedings of FJCC, Vol. 41, part I, 1972, pp. 587-596.
14. PARHAMI, B., "A Highly Parallel Computer System for Information Retrieval," Proceedings of FJCC, Vol. 41, part I, 1972, pp. 681-690.
15. PARKER, J.L., "A Logic per Track Retrieval System," IFIP Congress, 1971, pp. 146-150.
16. SAVITT, D.A., LOVE, H.H., TROOP, R.E., "ASP: A New Concept in Language and Machine Organization," Hughes Aircraft Technical Report No. TR-66-174 (AD-488538), June, 1966.
17. SU, S.Y.W., COPELAND, G.P., and LIPOVSKI, G.J., "Retrieval Operations and Data Representations in a Context-addressed Disc System," Proceedings of ACM Programming Languages and Information Retrieval Interface Meeting, Nov. 1973.
18. WANG, C.P. and LUM, V.Y., "Quantitative Evaluation of Design Tradeoffs in File Systems," Proceedings of the Symposium on Information Storage and Retrieval, April, 1971.

# DERIVING DESIGN GUIDELINES FOR DIAGNOSABLE COMPUTER SYSTEMS

John M. Hemphill, USAF  
S. A. Szygenda  
University of Texas

## ABSTRACT

Diagnosable computer systems are designed to detect and isolate the faults that occur during system operation. A number of techniques are available to the system designer of diagnosable systems. This paper examines a number of these techniques and derives a set of design guidelines incorporating them.

## I. INTRODUCTION

There are many reasons for needing design guidelines before one attempts to design a computer system. Perhaps the most important reason is the nature of the design process itself. Usually from a rough initial specification the design proceeds through a series of iterations between software and hardware designers. Eventually a prototype machine is actually constructed. Only then are many serious problems associated with hardware and software discovered. Usually the bulk of the effort directed toward developing diagnostic software and procedures does not occur until the machine is physically realized. Once the machine exists, it is difficult, if not impossible, to significantly alter it from its original design. As a consequence of the design process, the designer must include diagnosability from the very inception of his work, if diagnosability is a desired quality in the finished system.

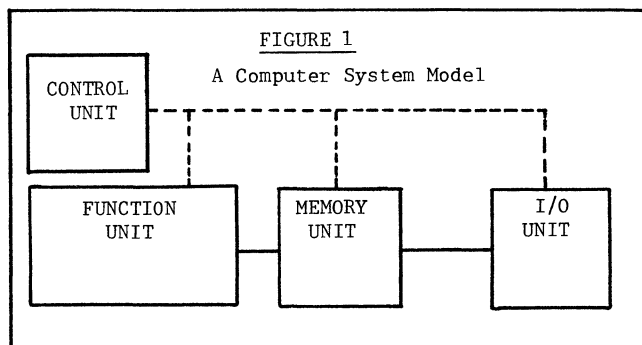
A diagnosable computer system is defined as one designed to detect the occurrence of faults before errors are introduced into the system operation and to aid in their isolation.

It is difficult to define what the scope of design guidelines should encompass. If design guidelines are too narrow and too specific, then they will only be applicable to a narrow class of systems. On the other hand, if the design guidelines are overly general, chances are they will be of little value. Perhaps the best goal is to attempt to develop design guidelines which can be related to physically realizable computer systems. Design guidelines hopefully should show where the system is sensitive to changes in design in order to facilitate obtaining the desired qualities in the completed system.

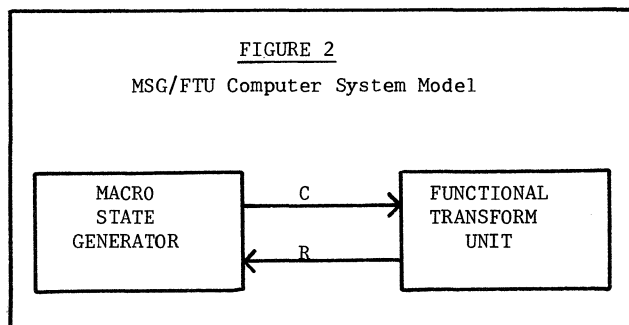
Supported in part by ONR-N00178-71-C0148,  
Naval Weapons Laboratory, Dahlgren, Virginia.

## II. THE MSG/FTU MODEL

In this paper, several design guidelines will be developed through the use of a computer system model. The design guidelines will be derived through the development of a number of conditions and relations pertaining to the operation of the computer system model. The model is general in nature and easily relatable to physically realizable computer systems. The operation of nearly any computer system can be simulated by an appropriate version of the model. In many cases, the structure of the model will be very similar to the actual structure of the computer system that it simulates. The model is developed from the common computer system representation illustrated in Figure 1.



The model is known as the macro state generator/functional transform unit (MSG/FTU) model and is illustrated in Figure 2. The macro state generator (MSG) is somewhat analogous to the control unit for the representation in Figure 1. The functional transform unit (FTU) is analogous to all elements in the computer system other than the control unit. Both the MSG and FTU are finite devices. Before using the MSG/FTU model as a vehicle for analysis, it is necessary to carefully define its mode of operation.



The MSG is connected to the FTU by means of two paths. The path labeled C in Figure 2 is known as the control path and is used by the MSG for transmitting data and command information to the FTU. The path labeled R in the same figure is known as the response path and is used by the MSG to obtain information pertaining to the operation and status of the FTU. It is important to remember that the action of a computer system is to simulate the operation of some virtual machine. Each virtual machine instruction is composed of more than one microsequence. A microsequence is the smallest basic operation that can be performed by the FTU. The MSG controls the execution of microsequences. That is, it regulates order and timing of the microsequences needed to compose the desired virtual machine instructions. The operation of the MSG proceeds on a discrete time base. That is, MSG operations can only be initiated at periodic points in time. These points are determined by the quantum time of the MSG. The quantum time or qtime is defined as the length of the minimum interval that can occur between the initiation of operations in the MSG at different points in time. All operations in the MSG occur at points in time which are multiples of the qtime periods. The qtime can be thought of as the basic clock cycle time in the MSG. The terminology "macro state generator" is derived from the fact that the MSG initiates operations in the FTU which can cause the FTU to transition through several true sequential machine states before the operation is completed. Hence, the modifier "macro" is added to indicate the scope of the state transitions.

The FTU contains the elements needed for the operation of the virtual machine. That is, it contains all functional units, memory elements, and data transfer paths used in the operation of the virtual machine.

Operations in the FTU are initiated by the control path from the MSG. Information concerning an FTU operation is transmitted to the MSG by the response path. The sequences of control information and response information transmitted between the MSG and FTU are known as the CR sequences. The complete set of CR sequences that can be executed by the MSG is known as the repertoire of the MSG. The only way that the virtual machine can obtain information about the status of the hardware is by the execution of CR sequences which can manipulate the memory and state of the FTU. Hence, the virtual machine program can then attempt to ascertain the state of the hardware by inspecting the memory and the state of virtual machine after execution of a virtual machine instruction.

### III. DEVELOPMENT OF THE MSG/FTU MODEL

The next step in the developing of design guidelines is to examine a series of conditions and relations pertaining to the MSG/FTU model. The first set of conditions to be considered deal with the nature of the information available about the operation of the FTU.

#### Condition 1

The MSG can only observe the operation of the FTU at a finite number of points.

Condition 1 is derived from the design of the MSG/FTU model. The MSG is finite. Additionally, the FTU is finite and only presents a finite number of observation points. Also, there are a number of unobservable points in the FTU such as values internal to switching elements.

#### Condition 2

The MSG can only observe the operation of

the FTU at discrete points in time.

Condition 2 is based upon the definition of the operation of the MSG. Any operation in the MSG can only occur at a point in time which is a multiple of the qtime of the MSG.

#### Condition 3

The MSG is the only element in the model that can observe the operation of the FTU directly.

Condition 3 is a result of the definition of how the model operates.

#### Condition 4

The set of FTU observations made by the MSG constitutes the only source of information concerning FTU operation available to virtual machine level programs.

The only manner in which a virtual machine level program can obtain information is through the manipulation of virtual machine memory and status by the execution of MSG microsequences.

By Condition 3, the MSG is the only element which can observe the FTU operations. Hence, the observations of the MSG are not only the sole source of FTU information, but the MSG initiated microsequences are the only means by which this information can be transmitted to the virtual machine level.

The key observation here is that information concerning the operation of the FTU is only available to a virtual machine level program through the auspices of the MSG. More over, the information available to a virtual machine level program is usually only a part of the information available to the MSG.

Much of the difficulty often encountered in attempting to perform a thorough diagnosis of a computer system is due to the design and complexity of the system. In diagnosing a computer system of some complexity, one is not just faced with attempting to diagnose a single combinational or single sequential logic system but rather a complex interconnection of combinational and sequential logic systems. Probably the most serious problem in performing diagnosis in a computer system is that of accessibility to the individual logic system being tested. Usually, it is not possible to access the individual combinational or sequential logic system by itself from the virtual machine level. Often, it is simply not possible to create effective and practical diagnostics from the virtual machine level due to the design of the system.

Condition 3 provides assistance in attempting to deal with the problem of system diagnosis. Condition 3 states that the MSG is the only element capable of observing the action of the FTU directly. By using the MSG, it is possible to utilize microsequences to perform diagnosis on the FTU. Diagnosis by use of microsequences is commonly known as microdiagnosis. Microdiagnostics have been employed for some time on a number of commercially available computer systems (1,2,4,5). Common arguments for the use of microdiagnostics are usually based on the fact that it gives greater access to individual logic systems in the computer. Also, use of microdiagnostics releases the diagnostician from having to build his diagnostic tests around the standard virtual machine instruction cycle of fetch instruction, fetch operand, and execute instruction. However, the use of microdiagnostics implies that the proper CR sequences either always exist in the repertoire of the MSG or that the repertoire of the MSG is not fixed and can be altered to meet the needs of the microdiagnostician. If the MSG of the system is

microprogrammable, then the repertoire of CR sequences is considered to be variable. An MSG with a fixed repertoire is known as a hardwired MSG.

Before proceeding to develop relations which are concerned with diagnosis of the system, the procedure of diagnosing the MSG/FTU system model will be examined.

In the case of either the use of a microprogrammed MSG or the use of a hardwired MSG, the MSG is the first logic system to be diagnosed because without assurance of the integrity of the MSG, no diagnosis of the FTU is possible. In the case of a hardwired MSG, diagnosis can present extremely serious problems. Normally, hardwired units do not possess a regular logical structure. As a result, external diagnosis by human diagnostic test input can be an extended project. Diagnosis of a microprogrammed MSG can be much more straightforward. Normally, a microprogrammed MSG unit possesses a much simpler and more regular logic structure than a hardwired MSG unit of similar ability. Additionally, since the repertoire of the MSG is variable, certain CR sequences can be included to aid the human diagnostic tester in performing diagnosis on the MSG. Once the integrity of the MSG has been verified, diagnosis of the FTU is the last step to be performed in diagnosis of the MSG/FTU system.

Diagnosis of the FTU can be performed by one of two different methods. First, diagnosis of the FTU can be performed through the use of virtual machine instructions which initiate a number of CR sequences determined by the type of virtual machine instruction. Second, individual CR sequences can be used to expressly diagnose portions of the FTU. In the case of a hardwired MSG, only the first method is available, while in the case of a microprogrammed MSG both methods can be utilized. Upon completion of FTU diagnosis, the system will be completely diagnosed.

In a hardwired MSG, one of the most difficult areas to diagnose is the circuitry used to generate the microsequences. Commonly, this is a microsequence selection matrix and associated timing sequence circuitry. For a background on the operation of such units, see Rosin (6). In actual practice, the control unit or MSG of systems is diagnosed by executing a virtual machine program. If the virtual machine program executes properly, the maintenance engineer assumes that the MSG or control unit is operating properly. Needless to say, this often is a self-defeating practice if the FTU contains faults. Unless some type of hardware tester is available to check the control sequences produced by the hardwired MSG, the diagnosis can be a tedious and difficult job. The inherent difficulty of diagnosing the hardwired MSG is one of several serious drawbacks to using the hardwired MSG in a diagnosable system.

Diagnosis of a microprogrammed MSG can be made a somewhat more feasible job than diagnosis of a similar hardwired MSG. This is due to the manner in which the microprogrammed MSG produces control sequences. The major component to be diagnosed is the control memory that stores the control sequences. Once the logic circuitry used to sequence the fetching and executing of control sequences from the control memory has been externally diagnosed, a program of diagnosing the control memory can be initiated. The type of diagnosis used for the control memory depends, of course, on the nature of the memory. If the memory is a read only memory such as transformer or capacitor read only storage, then it is likely that a section of control memory containing diagnostic test patterns will be inserted into the control memory unit. Once this is done, a diagnostic sequence can be run using the logic in the MSG to read the contents of the memory to check

for proper memory operation. Needless to say, this entails additional logic circuitry to provide for executing such diagnostic sequences and a method of verifying the diagnostic control memory. If the control memory is both a read and write memory, then a diagnostic sequence can be constructed to both write and read data to the control memory. Here again, additional logic circuitry may be required to facilitate implementation of the diagnostic sequences. Almost any advantage in diagnosis that the microprogrammed MSG has over the hardwired MSG is due to the more regular structure of the microprogrammed MSG. Even though the structure of the microprogrammable MSG is regular, it is very flexible in that it is programmable and as such retains the inherent power of a stored program computer.

The next step is to develop four relations which illustrate the differences between a hardwired and a microprogrammed MSG. The goal of these relations is to lend substance to the claim that a microprogrammed MSG is needed in diagnosable computer system environments.

#### Relation 1

Given: (1) an MSG with a fixed repertoire of CR sequences; (2) a specified virtual machine instruction set.

Then, the effectiveness of a diagnosis of the FTU by a virtual machine diagnosis program is sensitive only to the design of the FTU.

The basis of Relation 1 is that the given conditions constrain the design. Since the virtual machine instruction set is specified and can command only a fixed set of CR sequences, the only way to make diagnosis more or less effective is in the design and organization of the FTU.

The major observation that should be made at this point is that if it is desired to have effective FTU diagnosis, then the initial system design is the critical element in the process of system production. Once a hardwired system is physically realized, the only way to improve diagnosis is to write better virtual machine diagnosis programs. It is often the case that due to system design little improvement can be achieved on the virtual machine level. In fact, due to the complexity of most systems, many problems are unknown during the design phase and are only discovered upon fabrication of the system. This can lead to extremely serious diagnosis problems commonly reflected in the statement that the "diagnostic programs are programs that execute correctly when no other programs can execute due to hardware faults".

#### Relation 2

Given: (1) an MSG with a variable repertoire of CR sequences; (2) a specified virtual machine instruction set.

Then, the diagnosis of the FTU is sensitive to the design of the FTU and to the CR sequences available in the MSG repertoire.

The basis of Relation 2 is that even though the virtual machine instruction set is specified, additional CR sequences can be added to aid in the process of FTU diagnosis. In this way, the design of the FTU is not the sole determining element as it is in the case of Relation 1.

The key point to observe here is that by having a variable repertoire of CR sequences in an MSG it may be possible to enhance diagnosis after the system is fabricated. This is not to say that the design of the FTU is not as critical as it is in the case of the hardwired MSG. The design of the FTU is still a major

factor in the effectiveness of its own diagnosis.

### Relation 3

Given: (1) MSG/FTU system A having an MSG with a variable repertoire of CR sequences; (2) MSG/FTU system B having an MSG with a fixed repertoire of CR sequences; (3) virtual machine instruction set C; (4) both system A and system B can execute only instruction set C; (5) both system A and system B possess identical FTU units.

Then, (1) the ability of FTU diagnosis in system A to detect FTU faults during the diagnosis procedure is greater than or equal to the ability of system B to perform the identical operation; (2) the ability of FTU diagnosis in system A to isolate FTU faults is greater than or equal to the ability of system B to perform the identical operation.

Both parts of Relation 3 are based upon the same concept. The sets of CR sequences that can be executed in system B is fixed by the virtual machine instruction set and the design of the FTU. Hence, system B can only perform fixed sets of CR sequences. In the course of performing the diagnosis procedure on the system B FTU it can possibly be that either fault detection or fault isolation or both can be improved by the addition of a CR sequence not available in system B. If no CR sequences can be added to improve system B diagnosis, then system A will have a level of diagnosability equal to system B.

Relation 3 shows that enhanced FTU diagnosis may possibly be achieved simply by use of an MSG unit having a variable CR sequence repertoire instead of an MSG unit having a fixed CR sequence repertoire.

### Relation 4

Given the conditions in Relation 3.

Then, the number of CR sequences used to perform diagnosis in system B is greater than or equal to the number of CR sequences used to perform FTU diagnosis in system A.

The proof of Relation 4 is based on the fact that since the CR sequences are fixed in system B, it is possible that unneeded CR sequences will be executed in the course of FTU diagnosis since every virtual machine instruction causes execution of a predetermined set of CR sequences. If there are unneeded CR sequences, these CR sequences can be omitted in the diagnosis of system A since the MSG repertoire is variable. Hence, the length of the diagnostic sequence in system B is greater than that of system A. If no CR sequences can be omitted, then the lengths of the diagnostic sequences in both systems are equal.

## IV. DESIGN GUIDELINES

To achieve the design goal of diagnosability it is necessary that the system designer always keep concurrent fault detection and fault isolation central in the system design. The next section deals with guidelines pertaining to the MSG. Throughout this section it should be remembered that the guidelines are for the purpose of achieving the design goal of diagnosability.

There are a number of important items to be considered in the designing of an MSG. The design guidelines are as follows:

1. The MSG should have a variable repertoire of CR sequences, i.e. it should be microprogrammed. Relations 1,2,3, and 4 illustrate the advantages that are available to the microprogrammed MSG in diagnosis of the FTU. These advantages cannot be ignored since diagnosis of the FTU is a key part of designing a

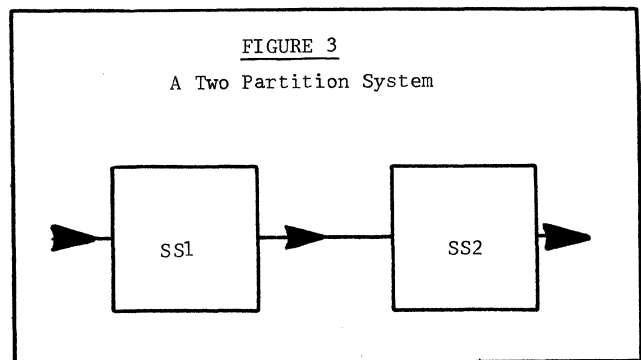
diagnosable computer system. This is not to imply that a hardwired MSG system is completely undiagnosable. The difficulty encountered with the hardwired MSG/virtual machine instruction diagnosis is that it implies attempting to diagnose the real machine by means of the virtual machine which itself is being simulated by the real machine. The process is somewhat self-defeating.

2. The MSG has to satisfy the diagnosable system design goals. That is, ideally all faults must be detected, and ideally all faults must be capable of being isolated. This goal is more easily approachable in a microprogrammed MSG due to its structure, which is more regular and often less complex than a comparable hardwired MSG. External fault diagnosis and isolation procedures for a microprogrammed MSG are more straightforward to develop than for a comparable hardwired MSG.

Many techniques exist for concurrent fault detection in a microprogrammed MSG. For example, the control memory could utilize fault detection techniques as proposed by Szygenda in his fault tolerant memory design(7). Coding techniques can be utilized to provide fault detection on gating function lines. Another important area of fault detection is in the area of timing sources used to sequence the operation of the MSG and FTU operations. A number of techniques for detecting faults in timing sources are available(3).

It is necessary to next consider the problem of FTU access for diagnosis. This is needed since the MSG must access the logic subsystems of the FTU in order to facilitate diagnostic testing of the FTU. Since, in any nontrivial system, diagnostic testing will be conducted on a non-monolithic or modular basis. It is important to consider the diagnosis of partitions and combinations of partitions.

Figure 3 illustrates a system composed of two partitions -  $SS_1$  and  $SS_2$ . If both  $SS_1$  and  $SS_2$  are strictly combinational logic, it is quite possible that access to only the input of partition  $SS_1$  and the output of partition  $SS_2$  will be sufficient for diagnostic testing and fault isolation. To determine if this is true, diagnostic test generation can be employed to discover how faults can be detected and how well they can be isolated. If either  $SS_1$  and  $SS_2$  or both are sequential logic systems, the picture changes and becomes less predictable. Problems arise in attempting to diagnose the composite system by means of using only the system input and the system output.



These problems arise from the nature of sequential logic systems. Sequential logic systems can be difficult to diagnose for three basic reasons. First, the sequential logic must be initialized to some known state for diagnosis. Whether or not this is a difficult task depends upon the design of the logic in question. If no means exists to preset the logic to a known state by using an additional input or preset line, then it is necessary to use a homing sequence to drive

the sequential logic to a known state before diagnosis can be started. Second, if the sequential logic is incompletely specified, there will be input sequences which can cause unpredictable outputs. Third, sequential logic can possess a large number of states which can make diagnostic testing a very extended task. For example, a 24 binary digit counter has over 16 million states. If the sequential logic is complex with a large number of possible states, diagnosis can be a difficult effort to accomplish. Combining sequential logic subsystems such as in Figure 3 can often result in a system that is more difficult to diagnose than its constituent subsystems. In most computer systems, usually more than just two subsystems are combined to obtain the system. Combining sequential logic subsystems results in a larger sequential logic system which presents correspondingly more difficult problems in diagnosis. Providing accessibility to the logic subsystems for the purpose of diagnostic testing is important. On computer systems not designed with sufficient access for diagnosis to logic subsystems, maintenance engineers often externally access logic subsystems with oscilloscopes and other instrumentation to facilitate diagnostic testing. Access to logic subsystems for diagnosis must be included in the initial system design.

#### V. CONCLUSIONS

The relations that have been presented in this paper illustrate that a microprogrammed control unit can have several important advantages in a diagnosable computer system. Namely, a microprogrammed control unit may enjoy enhanced fault isolation and a shorter diagnostic test sequence over a comparable hardwired control unit. Integrally enmeshed in the design of a diagnosable computer system is the need for access to the logic subsystems of the FTU for the purposes of diagnostic testing by the MSG. Design of a diagnosable computer system is a process of balancing access to logic subsystems with the corresponding ability of the MSG to properly use available FTU access points for effective diagnostic testing. To achieve a diagnosable system, the designer must keep this balancing process in mind in all of his design activities.

#### BIBLIOGRAPHY

1. Bartow, N., and McGuire, R. "System/360 Model 85 Microdiagnostics." AFIPS Conference Proceedings. Proceedings of the 1970 Spring Joint Computer Conference. Montvale, N.J.: AFIPS Press, 1970.
2. Guffin, R.M. "Microdiagnostics for the Standard Computer MLP-900 Processor." IEEE Transactions on Computers, Vol. C-20, No. 7, July, 1971, p. 803.
3. Hemphill, J.M. "The Development of Procedures for the Analysis and Synthesis of a Highly Defined Class of Fault Tolerant Computer Systems," Ph.D. Dissertation, August, 1971, Southern Methodist University.
4. Husson, S.S. Microprogramming Principles and Practice. Englewood Cliffs, N.J.: Prentice-Hall, 1970.
5. Johnson, A.M. "The Microdiagnostics for the IBM System 360 Model 30." IEEE Transactions on Computers, Vol. C-20, No. 7, July, 1971, p. 798.
6. Rosin, R.F. "Contemporary Concepts of Microprogramming and Emulation." Computing Surveys, Vol. 1, No. 4, December, 1969, p. 197.
7. Szygenda, S.A., and Flynn, M.J. "Coding Techniques for Failure Recovery in a Distributive Modular Memory Organization." AFIPS Conference Proceedings. Proceedings of the 1971 Spring Joint Computer Conference, Montvale, N.J.: AFIPS Press, 1971.





# DESIGN OF FAULT-TOLERANT ASSOCIATIVE PROCESSORS\*

Behrooz Parhami  
Algirdas Avizienis  
*Computer Science Department  
University of California, Los Angeles*

## ABSTRACT

Recent advances in computer technology have made the design of large and very flexible associative processors possible. Such systems are extremely complex and must be adequately protected against failures if they are to be used in critical application areas such as air traffic control or for performing control functions in fault-tolerant computers. This paper summarizes the results of a study which has indicated the techniques that are applicable in the design of fault-tolerant associative processors. Associative processors are divided into four classes of fully parallel, bit-serial, word-serial, and block-oriented systems. A technique for modularizing the design of an associative processor is given. The detection of errors within modules is discussed for the four classes mentioned above. Several schemes for reconfiguration are discussed which allow us to establish an appropriate intercommunication pattern after replacing the faulty module by a spare. The design of a fault-tolerant associative processor, which uses some of the techniques discussed previously, is presented.

## BACKGROUND

Associative processors are of interest since they enable us to solve many data processing problems for which digital computers with conventional architectures are either unsuitable or highly inefficient. Based on the applications that have been proposed for associative processors, there are at least two reasons for studying the fault tolerance problems of such devices: (1) In some proposed application areas, such as air traffic control [1], the effect of an undetected fault-induced error may be catastrophic. (2) To be able to perform control functions [2] in a fault-tolerant computer, an associative device must itself be fault-tolerant, since, otherwise, it will become part of the system's hard core and will contribute heavily to its unreliability. In addition, the extreme complexity of large, general-purpose associative processors necessitates the incorporation of fault tolerance features into their design.

It is remarkable, therefore, that the problem of fault-tolerance of associative devices has remained virtually untouched. Ewing and Davies [3] give techniques for coping with some hardware malfunctions in a plated-wire implementation of a particular associative processor. Proudman [4] suggests that a single error

\*This research was supported by the U.S. National Science Foundation under Grant No. GJ 33007X

correcting code can be used in conjunction with mismatch detectors with a threshold of 2 to detect storage errors. This paper summarizes the results of a study on fault tolerance techniques for associative processors [5]. We will concern ourselves with hardware faults and will assume the programs to be correct representations of intended algorithms for the specified domain of operation. We may note, however, that the simplified software of associative processors (e.g. fewer loops) with respect to conventional systems, results in a proportional simplification in the problem of software fault tolerance.

In the remainder of this paper, we will refer to fully parallel, bit-serial, word-serial, and block-oriented architectures for associative processors. This classification, which is based on the degree of parallelism in operations or, alternatively, the amount of storage associated with each unit of processing logic, is described briefly as follows. A more detailed discussion of these concepts and a comprehensive set of references can be found in [6].

- (1) In fully parallel associative processors, processing logic is associated with each bit of stored data. Most fully parallel systems implement only the exact-match search operation in hardware and use software techniques for arithmetic, logic, and more complex searches.
- (2) In bit-serial associative processors, processing logic is associated with each word of stored data. All the words can be processed in parallel, each in a bit-serial manner.
- (3) In word-serial associative processors, a single processing unit operates serially on all the words. This approach essentially represents hardware implementation of a simple program loop which is used for linear search.
- (4) In block-oriented associative processors, one block of information is associated with a unit of processing logic. A low-cost implementation of such a system may use a head-per-track magnetic recording memory in which each block is stored on one or more tracks.

## FAULT TOLERANCE APPROACH

Figure 1 shows a model for an associative processor which applies to the three classes of fully parallel, bit-serial, and block-oriented systems. Since word-serial associative processors closely resemble conventional systems, their fault tolerance problems can be studied separately. Each processing element (PE) in

Figure 1 consists of one unit of processing logic and its associated storage elements. In general, the processing elements in the PE array communicate with each other and the exact pattern of intercommunication is application-dependent.

A study of fault-induced errors in an associative processor shows that they are not easily detectable since a single fault may cause an arbitrary number of errors. This is evident for faults in global subsystems of Figure 1, such as the input and mask registers. A single fault in one processing element may cause errors in others because of PE intercommunication. The problem is further compounded by the fact that each PE performs logic and selective write operations on individual data bits which as we know are not easily checkable without a high level of redundancy.

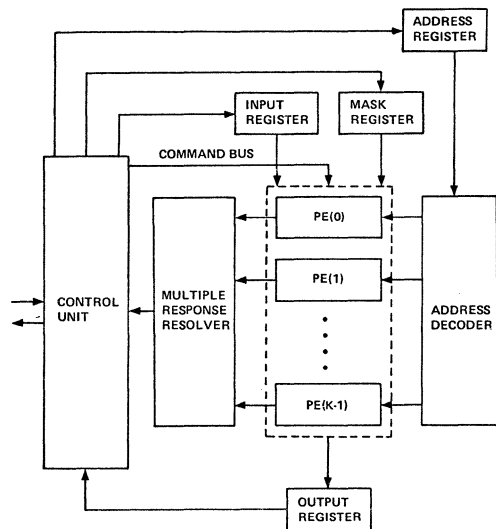


Figure 1. Associative Processor Model

The associative processor of Figure 1 can be made fault tolerant by dividing the PE array into identical modules which share spares. Let us assume that we have  $M$  modules, each consisting of  $P$  processing elements. It is possible to distribute the decoding and response resolution functions among the modules in order to reduce the complexity of the non-array portion to a minimum. Figure 2 shows the modules and their interconnections. One-dimensional intercommunication between modules has been assumed for simplicity.

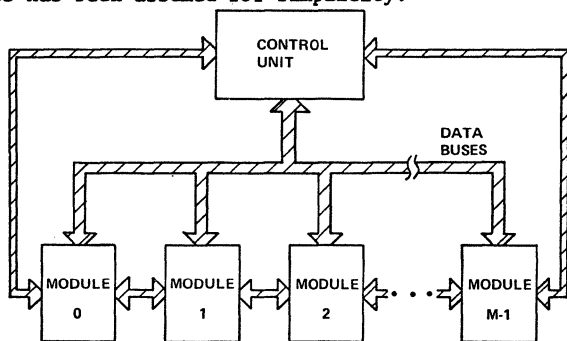


Figure 2. Modularized Associative Processor

Given a modular associative device as shown in Figure 2, it can be made fault tolerant by the following steps: (1) Incorporating internal failure detection ability within each module; (2) Adding  $S$  spare modules; and (3) Designing switching mechanisms and corresponding algorithms for reconfiguration. We will assume that the  $M + S$  operating and spare modules are permanently connected to the main data buses and that special isolating circuits exist between each module and

the data buses. Therefore, reconfiguration takes place by "power switching" and by providing alternate intercommunication paths between modules.

#### DETECTION OF MODULE FAILURES

We first discuss the problem of error detection in associative processors with respect to the four classes mentioned previously. Then we will consider a technique which is applicable in all cases.

A fully parallel associative memory with only exact-match search operation and without masking capability can be protected against storage errors by using a code with a minimum distance of  $k$  in conjunction with mismatch detectors with a threshold of  $k$ . With this scheme, stored words containing  $k-1$  or fewer errors will never respond to a search operation and are effectively isolated from the rest of the system until periodic diagnosis routines detect their failure. The difficulty is that such an associative device will have no application besides simple table look-up. For most other applications, masking capability, more complex search types, and arithmetic operations are essential.

Considerations for bit-serial systems are similar to those for fully parallel systems. One advantage which exists here is the serial processing of bits in each word. This allows us to artificially extend each operation to the entire word by performing "null" operation on bit positions not originally specified. Now, since all the bits of each word are processed serially, codes with low-cost serial encoding and decoding can be used to protect against storage errors. It should be noted, however, that if operations on small fields within the words are to be performed frequently, the above scheme may result in a significant reduction of speed.

As noted earlier, because processing is performed serially in a word-serial system, protection against failures becomes relatively simple. Low-redundancy coding can be used to protect against storage errors. Failures in the processing logic may be detected through self-checking [7] design. Self-checking translators may be needed to convert the storage encoding ( $S$ -encoding) to an encoding suitable for processing ( $P$ -encoding). The main requirement on the  $P$  and  $S$  encodings is that fast (parallel) translation between the two must be possible.

One favorable property of block-oriented systems with respect to fault tolerance is that during each operation cycle, a processing element operates on the entire block of information assigned to it. This enables the use of block codes which result in relatively low redundancy and have simple serial checking algorithms. If mechanical storage devices are used to implement such devices, error bursts become very probable due to dust particles, minute scratches, or defects in the oxide coating. It has been noted that low-cost arithmetic error codes are very effective for coping with such burst errors [8].

As can be seen from the previous discussion, low-redundancy coding techniques are applicable only in special cases. Design of logic circuits in self-checking form [7] (i.e., in a way that internal circuit failures manifest themselves on the circuit's output) particularly if 1-out-of-2 encoding is used, appears to be promising. However, because of the relatively higher complexity of the self-checking design approach as compared to low-redundancy coding techniques, this approach should be used when others fail or for protecting the system's hard core. A detailed discussion of self-checking design concepts is beyond the scope of this paper [5].

#### RECONFIGURATION THROUGH SWITCHING

For a modular associative device to tolerate module failures, the module interconnections should not be rigid as shown in Figure 2. Rather, the modules should be interconnected through specially designed switching cir-

cuits which prevent a system failure as a result of the failure of a module. The setting of these switching mechanisms determines the system configuration and can be changed by a central monitor if required. If a module error is indicated and the existence of a permanent failure is determined, reconfiguration procedures must be initiated to establish a new working configuration. In general, data transfers between modules and correction of fault-induced errors are needed as part of the reconfiguration process

We will assume only unidirectional (left to right) data flow between the modules in Figure 2. The generalization of the results to bidirectional data exchange is straightforward. After detecting the existence of a faulty module, the following steps must be taken before normal operation can resume: (1) Locating the faulty module; (2) Determining a new working configuration; (3) Initiating appropriate data transfers; and (4) Effecting reconfiguration through switching. The criteria that should be used in evaluating each reconfiguration scheme include: (1) The amount of data transfers needed; (2) The complexity of the reconfiguration algorithms; (3) The number of spares  $S$  needed for tolerating  $f$  module failures; and (4) The complexity of additional switching circuitry.

We first discuss centralized reconfiguration schemes in which the switching hardware is external to the modules. A straightforward solution is the use of a "permutation network" [9] which can interconnect the modules in any order. Such a permutation network can be implemented as a cellular array [10] of two-state basic modules. Since the complexity of such a cellular permutation network is roughly proportional to the square of the number of modules, its use can be justified only if a relatively small number of modules are involved. The two-state basic modules can be used in a different way to form a "shorting network" [9]. As shown in Figure 3, such a shorting network can be used to route data around the faulty and spare modules. One disadvantage of this scheme, particularly as shown in Figure 3, is the excessive amount of data transfers needed in the case of a failure. The number of transfers needed can be reduced by optimal placement of the spare modules [5].

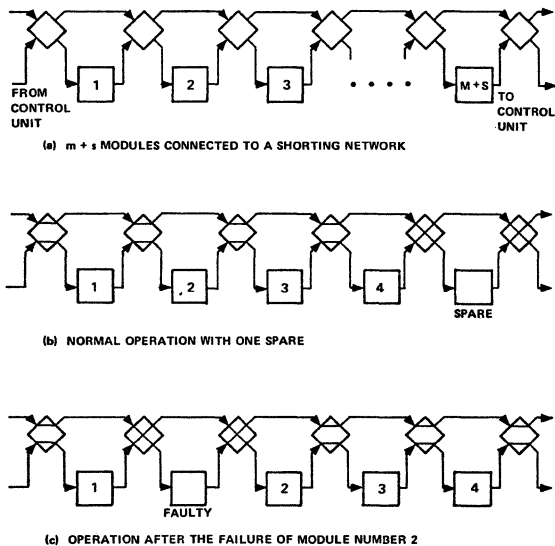


Figure 3. Reconfiguration with a Shorting Network

Another approach to the reconfiguration problem is the use of a distributed switching mechanism; i.e., distributing the switching hardware among the modules. This can be done by providing each module with a set of input and output lines instead of one as shown in Figure 2. Then if a successor module connected to one module output fails, a module connected to another output can act

as its successor. The simplest case, which will be discussed here, is when each module has two sets of inputs and two sets of outputs. The two inputs and two outputs are distinguished by the letters H and V (horizontal and vertical). The module has four states denoted by HH, HV, VH, and VV, depending on whether the H or V input is used and whether the output is generated on the H or V output.

Figure 4 shows a two-dimensional arrangement of the basic modules. It can be seen in Figure 4 that the 9 modules can be connected into a string. If any single module fails, the remaining 8 can continue their operation. Double module failures will leave at least 6 usable modules. Hence, with  $M=8$  and  $S=1$ , this scheme can tolerate all single module failures. With  $M=6$  and  $S=3$ , all double failures can also be tolerated as well as some triple failures. The problem of optimal interconnection patterns for the tolerance of a maximum number of module failures has not been solved. The basic advantage of this scheme is that the switching mechanism is not part of the system's hard core since a failure in the switching circuits is equivalent to a module failure. The main disadvantages of this scheme are the complexity of the reconfiguration algorithm, excessive data transfers, and tolerance of fewer than  $S$  failures.

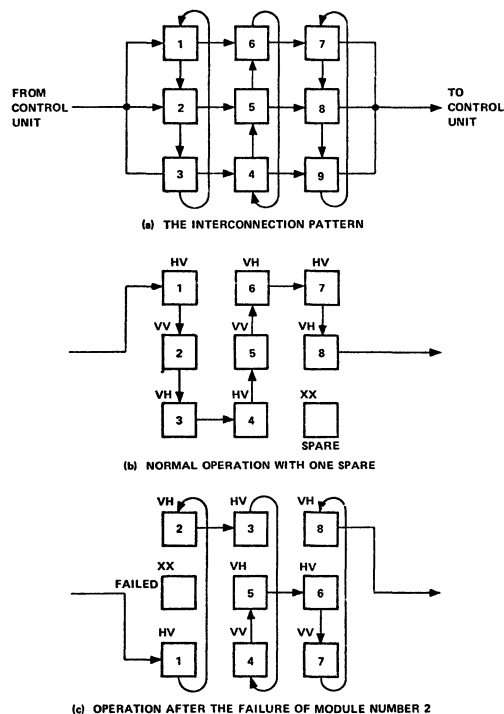


Figure 4. An Example of Distributed Reconfiguration

### A CASE STUDY

In this section, we illustrate the applicability of some of the techniques discussed previously by presenting the design and evaluation of a fault-tolerant associative processor called SPARE (inverse acronym for Error-tolerant and Reconfigurable Associative Processor with Self-repair). SPARE is essentially a fault-tolerant version of an associative processor which has been described previously [3]. Figure 5 shows a block diagram of the non-redundant system. The random-access memory is used for storing instructions and constants and consists of 4096 24-bit words. The associative memory contains 512 96-bit words.

The non-redundant associative processor of Figure 5 can be divided into two parts: (1) The associative (parallel) section, which consists of the associative memory array, bit column selection logic, and word logic; (2) The control and sequencing (sequential) section,

which contains all other subsystems of Figure 5. The sequential section uses status signals and test inputs for monitoring the operation of the parallel section. We now briefly discuss the three main features of SPARE; i.e., error tolerance, reconfigurability, and self-repair.

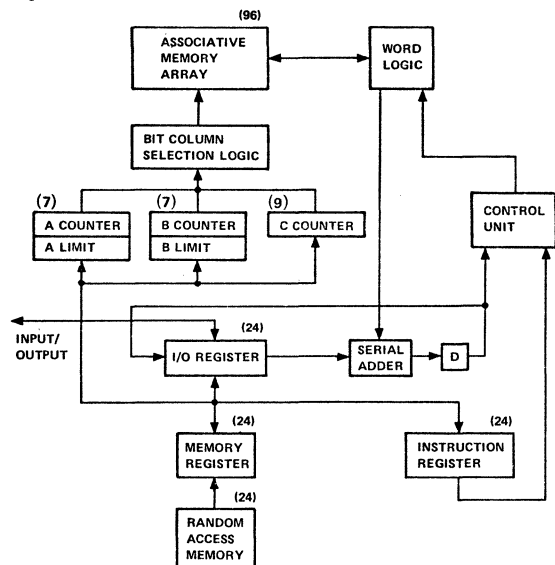


Figure 5. The Non-Redundant Associative Processor

To achieve error tolerance, the parallel section of SPARE is divided into  $M$  identical modules.  $S$  spare modules are shared by the operating modules. Each module has internal failure detection capability which is provided by self-checking design of its circuitry using two-rail encoding of logic variables. When a module error is indicated to the sequential section, the recovery mode is entered and the final result may be the replacement of the faulty module by a spare module. The sequential section of SPARE resembles a small general-purpose computer and can, therefore, be made fault tolerant by conventional techniques.

One of the very important properties of associative processors is simple modular growth. The size of an associative processor can grow without a need to alter its algorithms. This suggests that if additional processing capability is required, the redundant processing logic in SPARE can be utilized. Even the two channels of the two-rail circuits can be used independently to double the processing capability if certain design criteria are met [5]. Specifically, we postulate the following operation strategy for SPARE: (1) During normal operation the system works in redundant mode with a number of spare modules; (2) If a module failure occurs or additional processing capability is needed and if a sufficient number of spares are available, they are switched in; (3) If a module failure occurs or additional processing capability is needed and spare modules are not available, the system reconfigures into simplex mode by utilizing the two channels of the two-rail circuits independently.

Of the reconfiguration techniques discussed in the previous section, the one using a permutation network seems to be suitable for SPARE since only one intercommunication line (two in self-checking design) exists between modules and the number of modules is expected to be small ( $M=4$  or  $8$ , for example). The self-repair process will then essentially consist of computing and setting of a new state for the permutation network. This process must be followed by a recovery procedure to transfer the data stored in the failed module to the one which replaces it. The permutation network has a two-rail self-checking design but no spare is provided for it.

In computing the reliability of SPARE, we will assume that the coverage factor  $C$  includes the reliability of the permutation network. Using the reliability modeling technique of Bouricius et al [11], we find the reliability improvement factor defined as  $[1-R_{nr}(T)] \div [1-R_r(T)]$  as a function of mission time  $T$  for several configurations of SPARE ( $R_{nr}$  and  $R_r$  denote the non-redundant and redundant reliabilities, respectively). Figure 6, which depicts the resulting curves, shows that for mission times which are short compared to the MTBF for the non-redundant system, a significant increase in reliability is possible with a low level of modularization and a relatively small number of spare modules.

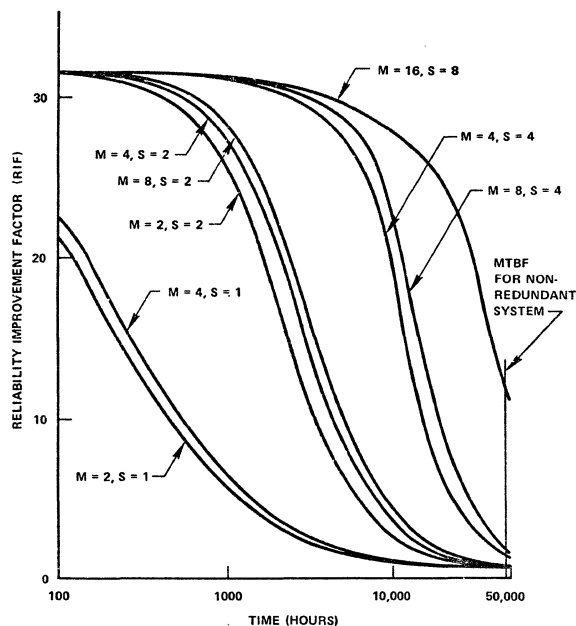


Figure 6. Reliability Improvement Factor for SPARE ( $C=0.99$ ).

#### CONCLUSION

In this paper, we have presented some results of a study on the fault tolerance of associative processors. Our main conclusions are as follows:

- (1) Dynamic redundancy is to be preferred over static approach because associative processors lend themselves naturally to modularization and since spares can be shared by a number of identical modules.
- (2) Low-redundancy coding techniques are applicable for error detection in associative processors but only in special cases. In particular, the use of arithmetic error codes for block-oriented systems appears to be promising.
- (3) Application of self-checking circuit design techniques seems to be an attractive alternative for error detection in associative devices.
- (4) Complex switching mechanisms and algorithms need to be devised to enable the sharing of spares by a collection of identical modules which communicate with each other.

Further research is needed in two equally important areas. The first area is the design of completely checked digital circuits. Systematic techniques need to be developed to aid the designers in choosing suitable input and output encodings and producing a self-checking design when presented with a non-redundant circuit or its functional behavior. The second area deals with general techniques for reconfiguration in array processors. Extension and generalization of the results presented here are possible in two directions. First, one can conceive of other interconnection schemes for

the case where one-dimensional intercommunication exists between modules. For example, we may consider a three-dimensional interconnection pattern in which there are three choices for each of the left and right neighbors for a module. Second, one may seek generalizations to the cases where multi-dimensional module intercommunication is used. This is a considerably more complex problem.

#### REFERENCES

- [1] Thurber, K.J., "An Associative Processor for Air Traffic Control," AFIPS Conference Proceedings, Vol. 38 (1971 Spring Joint Computer Conference), AFIPS Press, Montvale, New Jersey, 1971, pp. 49-59.
- [2] Berg, R.O. and M.D. Johnson, "An Associative Memory for Executive Control Functions in an Advanced Avionics Computer System," Proceedings of IEEE International Computer Group Conference, June 1970, pp. 336-342.
- [3] Ewing, R.G. and P.M. Davies, "An Associative Processor," AFIPS Conference Proceedings, Vol. 26 (1964 Fall Joint Computer Conference), Spartan Books, Baltimore, Maryland, 1964, pp. 147-158.
- [4] Proudman, A., "Bulk Associative Memory with Error Correction," IBM Technical Disclosure Bulletin, Vol. 12, No. 7, pp. 1076-1077, December 1969.
- [5] Parhami, B., "Design Techniques for Associative Memories and Processors," Technical Report UCLA-ENG-7321, Computer Science Department, University of California, Los Angeles, March 1972. (Also published as a Ph.D. dissertation).
- [6] Parhami, B., "Associative Memories and Processors: An Overview and Selected Bibliography," Proceedings of the IEEE, Vol. 61, No. 6, pp. 722-730, June 1973.
- [7] Carter, W.C. and P.R. Schneider, "Design of Dynamically Checked Computers," Information Processing 68, (Proceedings of IFIP Congress, Edinburgh, Scotland, August 1968), North Holland Publishing Company, Amsterdam, 1969, pp. 878-883.
- [8] Parhami, B. and A. Avižienis, "Application of Arithmetic Error Codes for Checking of Mass Memories," Digest of International Symposium on Fault-Tolerant Computing, Palo Alto, California, June 1973, pp. 47-51.
- [9] Levitt, K.N., M.W. Green, and J. Goldberg, "A Study of Data Commutation Problems in a Self-Repairable Multiprocessor," AFIPS Conference Proceedings, Vol. 32 (1968 Spring Joint Computer Conference), Thompson Book Company, Washington, D.C., 1968, pp. 515-527.
- [10] Kautz, W.H., K.N. Levitt, and A. Waksman, "Cellular Interconnection Arrays," IEEE Transactions on Computers, Vol. C-17, No. 5, pp. 443-451, May 1968.
- [11] Bouricius, W.G., W.C. Carter, and P.R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," Proceedings of the 24th National Conference of ACM, San Francisco, California, August 1969, pp. 295-309.



# A FAULT TOLERANT MULTIPROCESSOR ARCHITECTURE FOR REAL-TIME CONTROL APPLICATIONS

M. A. Fischler  
O. Firschein

Lockheed Palo Alto Research Laboratory  
Palo Alto, California

## ABSTRACT

This paper presents a fault tolerant multiprocessor architecture suitable for real time control applications requiring an extremely high degree of reliability. The architecture satisfies the following requirements:

- 1) Ability to deal with software as well as hardware faults: The proposed architecture is based on the assignment of distinct but redundant software modules to each task.
- 2) Efficient use of resources: The proposed architecture is a multiprocessor using time redundancy for fault correction. Thus, redundancy (beyond that needed for fault detection) is invoked only when a fault is detected. In normal operation, this extra capacity is available as an additional computing resource.
- 3) No hard core: In addition to the usual replication of system components, a partitioned system executive and a unique communication facility is defined which insures that the available redundancy will not be lost through a "domino" effect.
- 4) Interaction of computing units with sensors and effectors: The manner in which system architecture must be responsive to the amount and type of redundancy provided by the sensors and effectors is shown.
- 5) Use of current technology: The proposed architecture is based on the use of currently available hardware for the major system components.

After a detailed description of the architecture and the method of system operation, the system is related to existing fault tolerant systems, and unique characteristics of the present design are indicated.

## I. INTRODUCTION

There are many commercial and military control applications for which the computer technology is currently available, but due to the dire consequences of failure, computer systems cannot directly be used. These applications usually involve control of systems in which human life may be at stake, such as fly-by-wire aircraft control, or automatic braking of a train or an automobile. The present paper is concerned with techniques for achieving a sufficient degree of reliability to make presently available computers applicable to such systems.

### PROBLEM DEFINITION

In this paper we are concerned with problems of computer control of real time processes under the following conditions: 1) Ultra-reliable system operation (probability of failure approaching zero), 2) Use of current technology and mainly off-the-shelf components and subsystems, 3) Realistic cost constraints (i.e., limited use of hardware redundancy), 4) Completely specified task environment: all operations and actions required by the system are known and can be factored into the design, 5) Sensor and effector redundancy is sufficient not to be a limiting factor.

### ARCHITECTURE CONCEPTS

The following architectural concepts, used in the design, are discussed in some detail in the balance of the paper:

- a) Use of multiple and distinct software modules to detect faults (including design and translation faults).
- b) Use of time redundancy to correct detected errors (time redundancy is efficient in that the redundant computation, beyond that needed for fault detection, need be invoked only after an error is detected; resources can be used productively under normal operation).
- c) Integrated consideration of sensors, computer and effectors.
- d) No hard core items: distributed and partitioned executive control; redundant hardware, software, information storage, sensors, effectors, communications, power, etc.
- e) No 'domino' effect: isolation via hardware restrictions on communication and control.

## UNIQUE ASPECTS OF THE PROPOSED ARCHITECTURE

We believe that the architecture presented in this paper is unique with respect to the following items:

- a) A non-interfering type of broadcast communication system. (A discussion of alternative types of inter-module communication systems is presented in Appendix A.)
- b) A system executive which is partitioned into identical, independent, autonomous units. (A discussion of alternative types of fault tolerant executives is presented in Appendix B.)
- c) The consideration of design and translation errors (as well as damage faults) in both hardware and software.

Fault tolerance is achieved by the use of redundancy; however, unless a suitable degree of isolation exists between the major system components, it is possible that a single failure can destroy a redundant system. Items (a) and (b) appear to offer an extremely high level of isolation at a low cost in both system and hardware complexity. The increasing use of LSI circuits in the construction of computer hardware, with the associated infeasibility of exhaustive testing of such complex units, increases the probability that a computing unit will have undetected design errors. In the case of software, it is common knowledge that the complexity of such programs also make their exhaustive testing impractical. Therefore, it is necessary to design fault-tolerant computers not only from the point of view of protection against future device damage failure; rather, it may have to be assumed that both hardware and software design and translation errors are present initially. Consideration of this class of errors appears to be lacking in previously published works dealing with fault tolerant architectures.

## RELATION TO OTHER FAULT TOLERANT SYSTEMS

Even though organized in a unique way, many of the goals, constraints, and concepts we invoke are common to other systems. These include (references are examples, this is not meant to be an exhaustive listing): a) Off the shelf major subsystems, Ref. 9, 2; b) Lack of hard core, Ref. 9, 7; c) Multiprocessor organization, Ref. 9, 7; d) Time redundancy, roll-back, Ref. 1, 7; e) Minimal special hardware for fault detection or correction, Ref. 9; f) Loose synchronization, no lock-step, Ref. 9, 1; g) Adjustable degree of fault tolerance. Comment: This item

appears to be common to almost all of the FT architectures.

A comparison of the present system with three other fault tolerant systems is given in Ref. 6.

## II. ARCHITECTURAL CONCEPT

In the following discussion, we will present an architectural concept rather than a complete system design; therefore, decisions concerning the "best" alternative for some of the more detailed structure will not be made here, but must be determined by the specific environment to which the system will be applied. The level of detail of the discussion will be limited to the following major system components:

- Processing Units (PU): (mini) computers containing computational capability, registers, and scratch-pad memory.
- Memory Units (MU): random access memory augmented by some minimal logic capability (to be described later). Each PU has a MU assigned to it.
- InterModule Communication System (IMCS): the bussing and special registers used for internal communication.
- Timing and Synchronization System (TSS): the conventions by which the system components coordinate their activities.
- Input/Output Processors (IOP): elementary processors which interface between the computing system and the sensors and effectors. Input functions include A/D conversion, multiplexing, and buffering. Output functions include voting, buffering, and D/A conversion.
- Work Schedule and Contingency Plan (WSCP): a complete schedule for performing the required tasks, including allocation of resources to the tasks, and a contingency plan for task performance under various fault conditions (i. e., loss of resources).
- System Executive Software (EXEC): primarily the functions of fault detection and system status evaluation, as well as implementation of the WSCP.
- Sensors (S)
- Effectors (E)
- Application Software (AS)

### GENERAL SYSTEM DESCRIPTION

The system consists of two or more (preferably) identical PU, each with its own (functionally) identical resident EXEC and copy of the WSCP. Each PU, MU, and IOP has an output bus (which only it can write on) which goes to a dedicated read-only register in every other PU, MU, and IOP. The set of all such dedicated read-only registers in a system component will be called its Communication Memory (CM).

In the basic configuration, the computing system operates in a 4-phase cycle. During Phase I (P1), each PU works on one or more of the tasks assigned to it by the WSCP. Each task which forms a subset

of the application software is produced in three or more distinct versions and each version is partitioned into segments which can be run in the P1 time interval. (Typically, the execution of such a segment will result in a control signal to be sent to one of the effectors in P4 of the cycle.) During P1, exactly two versions of each task are executed on separate PU, and for this reason, the simplest arrangement would be an even number of PU grouped into pairs, with each pair of PU performing the same set of tasks.

During P2, paired PU compare results (partial or complete) via the IMCS. If an error is detected (difference in results), a HELP request is transmitted to the other PU of the system; otherwise, the valid data can either be saved for further processing, stored in the MU's assigned to the PU's, or stored in an IOP for later output to an effector.

During P3, a PU designated by the WSCP will respond to a HELP request by executing additional versions of the questionable computation. If no HELP requests are outstanding, low output rate, background, high-output rate (but not fully protected), self-check, housekeeping, etc., jobs are performed as specified by the WSCP.

During P4, HELP request situations are resolved by majority (or plurality) vote. This decision process is carried out in every PU of the system by comparing the data available in its CM. Defective system components are identified by their lack of agreement with majority, and their status is recorded in every PU. This up-dated status information, in conjunction with the WSCP, determines new job assignments and resource allocations within the system. Other activities carried out in P4 include acceptance, verification, and storage of sensor data and transmission of control signals to the effectors.

In the following subsections, we will expand and clarify many of the concepts presented in the above general system description.

### THE INTERMODULE COMMUNICATION SYSTEM (IMCS)

The IMCS aids in the performance of four primary system functions; these are: (a) Fault detection. It provides the means by which the PU can compare results. (b) Input/Output Communication. (c) Reconfiguration. In the event of failure of one of the physical units of the system, the communication patterns between the remaining units will typically change; the IMCS provides a simple means for accomplishing this function. (d) Isolation. The IMCS provides communication without allowing a defective unit to damage either other functioning units, or the IMCS itself.

As described earlier, the IMCS is implemented by having a (possibly redundant) bus assigned to each physical system module which only the assigned module can write on. The bus, simultaneously, drives a dedicated read-only register in every physical system module. The receiving module, based on its assigned duties as specified by the WSCP, will typically be attentive to only one of the registers in its CM at any given time, and ignore the remaining registers. Relevant multi-word messages are saved by the receiving unit reading and internally storing the incoming information; ignored data is overwritten and lost.

In a complete system design, a number of decisions concerning the IMCS and involving cost/performance trade-offs must be made. These include the width of the busses (i. e., serial or parallel data transmission),

\*Some copies of this report are available from the authors.



the possible use of error detecting or correcting codes to protect the transmitted information, redundant registers in a CM (or even redundant CM's in each unit), etc. While these decisions have important practical significance, they do not significantly alter the architectural design in a logical sense.

#### THE SYSTEM EXECUTIVE SOFTWARE (EXEC)

The EXEC, a duplicate copy of which is resident in each PU, performs three primary system functions; these are:

(a) Fault detection. This is accomplished in P2 of the basic system timing cycle by comparing the outputs of the P1 application computations produced in the resident PU with those produced in the "paired" PU. This comparison is enabled by having each PU broadcast its P1 outputs over the IMCS during P2. When the comparison yields out-of-tolerance results (tolerance limits are supplied by the WSCP for each application module as part of the descriptive information associated with the module), the EXEC signals the detection of a fault by broadcasting a HELP message to the system via the IMCS.

(b) Status determination; a recording, in each PU, of the proper or improper functioning of all system units. In P3, following the issuance of a HELP request, the original computations are repeated (original PU's, and software, unless resources are reduced to the point that status determination is no longer feasible) together with the running of one or more additional versions of the software on additional PU's. In P4, the fault is resolved by majority (or plurality) vote as all PU's issue their results on the IMCS. In addition, each PU can also make a determination as to the source of the fault (localization to a PU/software module combination). If, in the repeated computations of P3, the fault disappears, a transient error is recorded against both the responsible PU and software module. If the fault is repeated, then at the earliest possible later time, as specified by the WSCP, the computation is repeated with the suspect software (and original data set) run in a PU other than the original one; a correct result now permits assignment of the fault to the original PU, an incorrect result causes assignment of the fault to the software module. Status determination for MU's and IOP's is also performed and will be discussed later.

(c) Implementation of the WSCP. Each PU has its own copy of the WSCP, and record of the status of the various system units which is compiled as described above. The WSCP is a complete description of the duties to be performed by each system unit, keyed to the operational condition (status) of the system resources. Thus, without the need for any additional coordination between system units, each EXEC schedules its portion of the system workload on its host PU.

The existence of a system EXEC in each PU does not rule out the presence of a complementary local executive which might be concerned with such functions as local fault isolation and recovery, interrupt processing, service calls, etc.

#### THE WORK SCHEDULE AND CONTINGENCY PLAN (WSCP)

As defined previously, the WSCP is a complete description of how the workload is to be carried out by the system units. Such a plan, of course, must be

custom-tailored to the specific application environment and its requirements. However, the question of how to treat units which have demonstrated faulty behavior is of a somewhat more general concern. The architectural philosophy we are advancing here is one which anticipates and is tolerant of occasional errors (including those caused by faulty design) in both hardware and software. Thus, unless a unit produces such a high incidence of failures that the timing commitments of the system are threatened, no redistribution of workload is required. As a precautionary measure, software modules showing faulty performance could be reloaded from permanent storage. Further, local diagnostic and recovery procedures (e.g., invoking redundant hardware within a PU) might improve the condition of a damaged hardware unit. In general, even when a unit is severely damaged, it should still be able to perform some functions correctly and thus contribute to verification tasks. That is, if a damaged unit "A" produces an answer to a computation which agrees with unit "B" but disagrees with unit "C", we would be inclined to accept the answer of "B" as correct. The architecture presented here is able to accept such marginal contributions by damaged units.

#### THE TIMING AND SYNCHRONIZATION SYSTEM (TSS)

The four phase basic system timing cycle has been functionally described in previous sections and we will restrict our discussion here to the questions of timing requirements and synchronization.

In a real-time control application, we have the requirement to provide control signals to the effectors at (typically) regular intervals. For full fault tolerant operation, the basic system timing cycle should have a duration less than the duration of the cycle corresponding to the highest rate control signal. A basic system cycle time of 10 ms. or larger appears suitable for a wide range of applications. This implies that hundreds of individual commands can be executed (phases 1 and 3) prior to result comparison (phases 2 and 4), with buffering of results prior to comparison employed as required. Coordination of activities in the various system units does not require a lock-step type of synchronization, and thus we do not require a precision timing system. In particular, it appears advisable to adjust the task completion time of the tasks to be somewhat less than the phase interval (P1 or P4) in which the tasks are performed. Tasks which are not completed in their designated time intervals are assumed to have failed.

The actual timing mechanism could either be a single fault-tolerant system clock (Ref. 6), or could be accomplished by each PU announcing over the IMCS the current system phase according to its own internal clock. A majority vote of these timing signals is used to determine the actual system phase time and the various PU's can be resynchronized accordingly. By allocating a small amount of tolerance (or dead) time to each phase, slower units should be able to complete their tasks and resynchronize without trouble.

During startup, or resynchronization after a catastrophic failure, each unit follows the procedures specified by the WSCP and announces its current perception of the system state over the IMCS, just as it would for any other task.

For some applications, outputs might be required at a rate faster than feasible for the basic timing cycle. In such a case, two alternatives are possible. Three (rather than the normal two) versions of the task can

be executed in parallel and conventional majority logic used to determine the output. Such special handling of a task would not interfere or conflict with normal system operation. The second alternative is to transmit the control signals at the required rate with no assurance of immediate correctness, but the existence of faulty output would be detected, and corrective action initiated within one basic system cycle time.

#### THE PROCESSING UNITS (PU)

These devices are assumed to be conventional, off-the-shelf (mini) computers, or even single chip computers, with the possibility of a few minor additions. In particular, each machine requires a CM whose registers can be individually loaded under external control. A hardware (multiargument) voting operation would be very desirable, though certainly not essential. Each PU has a small amount of internal working storage capacity (scratch pad memory).

#### THE MEMORY UNIT (MU)

The MU's are random access storage devices which have the primary responsibility for verifying and maintaining the application data bases; that is, the sensor data and precomputed information needed to calculate the new effector control signals. Thus, if a disagreement is detected in the computed outputs of paired PU's, and a new PU is assigned the task of resolving this disagreement, the new PU can access one of the relevant MU's to obtain the data appropriate to its computational task.

Like all other major system units, the MU has a CM and some simple logic which includes the ability to compare and vote on multiple data items. During normal operation, a MU is assigned to a single PU and obeys its storage (only during P2 and P4) and retrieval commands (at any time) by paying attention to the appropriate register in its CM. It also intercepts, votes on, and stores data from sensors relevant to the tasks assigned to its associated PU. If the sensor data is not consistent, it stores the majority opinion and tallies the disagreement for later fault reporting. Before storing data from its own PU (i. e., permanently altering the data base), it compares this data with that produced by the paired PU; if there is a disagreement, no storage takes place until after the resolution of the resulting HELP request. During the HELP procedure, it responds to information requests from other PU's assigned to the failed task. In P4, following the help request, it determines and stores the majority opinion.

A MU can be reassigned to another PU if its own PU is judged to be inoperative by a majority vote of the PU's in the system. It can also be used in a shared mode (i. e., it now services more than one PU) when so instructed by either its own PU, or by a majority vote of the system PU's.

We note that the data base associated with each task is normally stored (in verified form) in two MU's. However, to be able to resolve a disagreement in the stored data should one of the MU's suffer a failure, we require that error detecting encoding be employed by the MU's.

#### THE INPUT/OUTPUT PROCESSORS (IOP)

Input functions such as signal sensing, A/D conversion, multiplexing, and buffering are usually required in interfacing a computing system to the real world. In a fault tolerant system, these functions must

satisfy the same criteria with respect to redundancy and isolation that we require of our computing components. Thus, we would expect that critical sensing devices be at least triply redundant, and their signals reach the computer through at least three independent processing (A/D conversion, multiplexing, and buffering) paths. We will call each such path an Input Processor, and assign it a register in all CM's of the system.

The normal output functions of a real time control system, including D/A conversion, demultiplexing, and buffering, as well as final effector activity, must again satisfy the fault tolerant criteria that is required of the rest of the system. In the case of the effectors, this consideration can be critical, since a single (non-redundant) effector performing a vital function can render useless all the prior redundancy built into the system. Thus, for each task which involves effector activity, we require at least triple redundancy. Some effectors can provide this capability in a single device by internally performing a voting operation (e. g., an actuator with multiple inputs which performs the voting hydraulically); however, even this capability is not satisfactory due to the lack of isolation in achieving the redundancy (i. e., this single device is still a hardcore item).

Given that we have at least three independent signal paths to three independent effectors for each output task, we now must drive each of these paths with the control signals produced by two or more independent PU's. An Output Processor is defined as a device which includes the signal path terminating in an effector, and which obtains its output information based on a vote of the relevant information appearing in its CM (i. e., either agreement of two data items in P2, or plurality of three or more items in P4).

#### THE APPLICATION SOFTWARE (AS)

The proposed architecture imposes a number of constraints and requirements on the AS; these include:

(a) The requirement for three or more distinct versions of the software for each application task (to help detect design and translation errors, as well as damage faults, in both hardware and software). While considerable attention has been devoted to determining whether two logical constructs are identical, the question of criteria for establishing degrees of distinctiveness does not appear to have been previously considered. For most cases of practical interest, we might assume that modules programmed by different programmers would satisfy the distinctness requirements. The topic of distinct software is discussed further below.

(b) The requirement for a common data base; that is, all software versions of each specific task must be able to use information stored in a common data base. This programming convention is necessary to allow a new version of a software module to be called into execution to resolve a conflict, without having to either maintain or generate a separate data base for such a standby software module. In a control application environment, where the data base would typically contain previously recorded sensor values and computed system states, a standard method of storing such data seems quite reasonable.

(c) The requirements for partitioning the software into segments which can be executed within the P1 time interval, and which will produce an (intermediate or final) output suitable for comparison and fault detection. The partitioning requirement is compatible

with the need for control signals at short periodic intervals typical of the control environment. Even in those cases where the computation must extend over many basic system cycles, and no reasonable intermediate results can be produced prior to completion of the computation, normal error protection can still be provided if the computation time interval (in system cycles) is a suitably small fraction of the required output cycle time. In this case the verifying computation will not be completed in a single P4 interval but will extend over a series of P4 portions of successive system cycles.

Distinct software. As noted above, an important aspect of the present fault tolerant design is the availability of distinct software, i.e., processors performing the same task must each have a program which satisfies the same specifications, but each program must use procedures that are "different" in some sense. Since identical processors are to be used in the system, the motivation for such distinct programs is that if identical processors are subject to the same fault situation, the programs (and hence the processors) will be in a different state when the fault occurs.

In converting from specification to procedures, distinct programs can be obtained by utilizing:

- 1) Different theoretical methods of converting specifications, e.g., using different methods for mechanizing the z-transform, or different methods of mechanizing trigonometric functions.
- 2) Different procedural methods, obtained either by using two or more persons writing programs based on the same specifications, or by the conscious inter-change of independent software procedures.

### III. SUMMARY AND CONCLUSIONS

In this paper, we have presented an architecture which satisfies the following requirements for real time control applications:

- 1) Ability to deal with software as well as hardware faults: The proposed architecture is based on the assignment of distinct but redundant software modules to each task. We have shown how communication, synchronization, and resource allocation can be handled at the system level to deal with the problems arising from such an approach.
- 2) Efficient use of resources: The proposed architecture is a multiprocessor using time redundancy for fault correction. Thus, redundancy (beyond the minimal requirement of duplicate computation needed for fault detection) is invoked only when a fault is detected. In normal operation, this extra capacity is available as an additional computing resource.
- 3) No hard core: In addition to the usual replication of system components, we have defined a distributed and partitioned system executive and a unique communication facility which insures that the available redundancy will not be lost through a "domino" effect. In particular, we have addressed and posed a solution to the question of a defective unit "locking-up" the communication channels, and bringing down the entire system.
- 4) Interaction of computing units with sensors and effectors: We have discussed how system architecture must be responsive to the amount and type of redundancy provided by the sensors and effectors.
- 5) Use of current technology: The proposed architecture is based on the use of currently available

hardware for the major system components. For example, the processing units will typically be conventional minicomputers or even single chip computers.

Since we have been primarily concerned with the logical organization of a fault tolerant architecture at a systems level, there are many questions we have not addressed. Thus, for example, the details of making the individual PU's internally fault tolerant has not been considered. We would assume that redundant power supplies are used, but have not discussed this point. Physical separation of the units and separation of the communication busses seems desirable, but was not discussed.

We raised the question of distinct but redundant application software modules with requirements for a standard data base, and rough synchronization of computation. While achieving these requirements for a particular application seems relatively straightforward, our continuing efforts will be directed to formalizing this process. Finally, we feel that the proposed architecture is suitable for a general problem environment as well as real time control applications; we plan to extend the simple supervisor presented here to permit extension to the general multiprocessor domain.

### BIBLIOGRAPHY

1. Avizienis, A. A., et al., "The STAR (Self-Testing-And-Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," IEEE Trans. on Computers, Vol. C-20, No. 11 (Nov. 1971)
2. Brosius, D. B. and Jurison, J., "Design of a Voter-Comparator Switch for Redundant Computer Modules," FTC/3 (See Ref. 3)
3. Daly, T. E., Tsou, H.S.E., Lewis, J. L., and Hollowich, M. E., "The Design and Verification of a Synchronous Executive for a Fault Tolerant System," Int. Symposium on Fault Tolerant Computing (FT/3), June 1973, Palo Alto, California (IEEE Cat. No. 73CH0772-4C)
4. Daly, W. M., Hopkins, A. L., and McKenna, J. F., "A Fault-Tolerant Digital Clocking System," FTC/3 (See Ref. 3)
5. Farber, D. J., et al., "The Distributed Computer System," Sixth Annual IEEE Computer Society Int. Conf. (Comcon '72), San Francisco, California, Sept. 1972 (IEEE Catalog No. 72CH0659-3C)
6. Fischler, M. A. and Firschein, O., "A Comparison of Fault Tolerance Concepts for Computer Architecture," Lockheed Missiles & Space Company Report, Oct. 1973
7. Hopkins, Jr., A. L., "A Fault-Tolerant Information Processing Concept for Space Vehicles," IEEE Trans. on Computers, Vol C-20, No. 11 (Nov. 1971)
8. Roberts, L. G. and Wessler, B. D., "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N. J.
9. Wensley, J. H., "SIFT-Software Implemented Fault Tolerance," AFIPS Conf. Proc., Vol. 41, Part II, 1972, Fall Joint Computer Conf., AFIPS Press, Montvale, N. J.

APPENDIX A

TECHNIQUES FOR INTERMODULE COMMUNICATION

Given a set of computing, memory, and I/O modules which are to contribute to the processing tasks of a system, there are two major factors which affect the communication between these modules:

- 1 - Connection characteristics, the nature and topology of the paths between the modules.
- 2 - Control mechanisms and communications protocols, the method of controlling the communication between modules and the nature of the communication.

Various types of topology and control commonly used are indicated in Table A-1, and examples of some of the communications systems used in fault tolerant designs is tabulated in Table A-2.

TABLE A-1

Types of Communication Interconnections, Control Mechanisms, and Protocols

- 1 - Connection characteristics
  - a - Topology
    - Each module connected to a central module
    - Each module connected to n other modules
    - Each module connected so as to form a ring
  - b - Nature of connection
    - Direct wire, module to module
    - Single or multiple common bus (uni- or bi-directional)
- 2 - Control mechanisms and protocols
  - Permission to send/receive given by central unit, or according to standard protocol (e.g., as in a conventional computer multiplexer bus).
  - Each module has specified time slot to send/receive.
  - "Lazy Susan", messages inserted into communication stream when empty slot appears.
  - Random transmit/receive.

TABLE A-2

Examples of Different Communication Approaches

ARPA Net  
(Ref. 8)

<u>Topology</u>	<u>Controls and Protocols</u>
Each station communicates with two or three other stations	No central control. Lazy Susan variation; packet of information addressed to destination with no specific routing indicated. Packet is forwarded from node to node until destination is reached.

UC Irvine Ring  
(Ref. 5)

<u>Topology</u>	<u>Controls and Protocols</u>
Ring	No central control. Lazy Susan arrangement with each processor examining the data stream as it goes by. Addressing is by process rather than destination.

SIFT, Stanford Research Institute  
(Ref. 9)

<u>Topology</u>	<u>Controls and Protocols</u>
Multiple common bus to each module; single dedicated line between each processor and its memory	Processors can read from any memory via common bus; processor can only write into its own memory.

STAR, JPL  
(Ref. 1)

<u>Topology</u>	<u>Controls and Protocols</u>
Common busses	Central control by TARP

TRW  
(Ref. 3)

<u>Topology</u>	<u>Controls and Protocols</u>
Each CPU and associated memory are connected to an input/output control unit (IOCU). Each IOCU is connected to a set of multiple data busses. External devices communicate with the computer via these busses.	Central system control unit monitors and controls communication.

Autonetics  
(Ref. 2)

<u>Topology</u>	<u>Controls and Protocols</u>
Four CPU's on multiple common data bus	VCS control to external output.

Proposed System

<u>Topology</u>	<u>Controls and Protocols</u>
Multiple common busses, one bus originating at each module, and leading to read-only registers in all other modules	Random transmit capability by each module on its own bus. Random receive/read capability by each unit with respect to its read-only registers which receive information from the multiple busses.

APPENDIX B  
TYPES OF SYSTEM EXECUTIVES

The executive of a fault tolerant system receives reports of system failure, controls the procedures that are to be followed when such failures occur, and re-allocates resources as required. The executive can take one of the following forms:

1 - Central executive

A central executive is one in which a single monolithic structure controls the overall system from a single PU. The two mechanizations possible are:

Hardware executive, e. g., a voter/comparator/switch (VCS) unit, whose logic design determines the procedures to be followed. This type of executive is usually limited to controlling system output based on majority vote of identical computations, switching out defective modules, and switching in spares. (Ref. 2)

Software executive. An executive mechanized in software is usually designed to handle a more complex set of situations. The software executive often deals with failures by changing the task allocation tables, so that defective modules are lightly loaded or ignored altogether. For reliability purposes, duplicate copies of the executive may be available in auxiliary storage, or even used to monitor the performance of the controlling executive. (Ref. 3)

2 - Distributed executive

In the distributed executive, an attempt is made to spread the executive functions for both efficient operation and so that in case of damage to one part of the system, executive capability will still be available. Two types of distributed executive are possible:

Multiprocessed executive. The tasks of the executive are carried out by more than one computer, and voting is then used on the multiple outputs. An example of this approach is the SIFT, Ref. 4.

Partitioned executive. The present design uses a partitioned executive in which each part of the system operates autonomously, based on observations of how the rest of the system is performing.

Discussion

In most of the multiprocessor approaches, if the executive encounters a design error which causes it to fail, the entire system can fail. For example, if a particular combination of tasks causes a "lock-up" condition, then the system will not be able to proceed past that state. In the case of the partitioned executive, however, each module is operating in its own task sequence under its own executive. If an executive failure should occur, only that processor is affected by the failure.



# A VARISTRUCTURED FAIL-SOFT CELLULAR COMPUTER

G. J. Lipovski  
*University of Florida*

## ABSTRACT

The architecture of a von-Neumann class computer is considered, in which the user programmer can request, at the beginning of his task, one of many word widths, and one of many memory heights. Several users are able to space share the computer. We call this feature varistructure. The computer is a minimally, yet strongly connected cellular structure consisting of microcomputers, and has the capability of being fail-soft.

## 1. INTRODUCTION

There is a growing desire for fail-soft computers. Especially where the computer performs an essential or very important function, it would be very desirable that the whole system need not stop working where one part fails. There is also a strong desire for a computer that is made of at most a few basic modules which are connected in a regular way, that is, a cellular computer, especially if the number of connections is minimal. If these objectives can be obtained for a computer that looks like a conventional von Neumann class computer to programmers, that computer should be quite effective! We will show a computer having these characteristics.

In this paper, we will consider the techniques used to support varistructure. In the next section we look at the cell interconnections. We consider the nature of the interconnections, the topology. Then we interpret this in terms of data transmission paths. In section 3, we examine the cell. We consider the construction of a suitable cell for our examples, general operation of an instruction cycle, and the meaning of STRUCTURE states. In section 4, we consider the operation of the cellular machine for the memorize of recall cycles. We describe the variable structure concept and the mechanism to select the height and width of memory. In section 5, we show how the execute cycle can be done. In particular, we consider the carry link operation in this machine. Section 6 shows the fetch cycle. Section 7 shows how the structure can be set up and section 8 gives our conclusions concerning this machine.

In order to explain the techniques used in this processor, we will arbitrarily choose an eight bit wide CPU and memory configuration. We will assume the address is sent on a separate link. We will also assume a standard one accumulator CPU structure. None of these assumptions are necessary for the architecture. In particular, there are many ways to time-share the links to decrease the number of pins, and so on. We would not propose building the machine in the way we describe

it. However, it is expedient to simply describe an example of this architecture so that the techniques are clearer. We choose the simplest example.

## 2. CELL INTERCONNECTIONS

### 2.1 Topology

The study of the interconnection of cells, the topology, is the key to minimal connected fail-soft computers. We propose to seek out the class of all structures with the property that, for a fixed number of nodes, there is a minimum number of (bidirectional) links such that the graph is strongly connected, and if any link is deleted, it is no longer strongly connected. All such graphs are trees! As we noted earlier (1), and as T. C. Chen also observed (2) the tree structure is fail-soft such that if any node is faulty, say an output driver is stuck-on-one, then the subtree can be pruned from the remainder of the tree, and the remainder can continue operating at reduced capacity. Since, in a homogeneous  $l$ -level tree with fixed fanout  $f$ , there are  $f^l$  leaf nodes,  $f-1$  nodes on the next higher level, and so on, almost all nodes are leaf nodes. For example, in a binary tree, half are leaf nodes, and in a ternary tree about 65% are leaf nodes. So a failure in most of the tree will cause small loss of performance because only a small subtree containing the failure will be extracted.

We also observe that a tree with fanout  $f$  having  $n$  nodes has delay approximately proportional to  $\log_f n$ . It is also possible to put the tree structure in a physical space in which the total delay is proportional to  $a\sqrt[3]{n} + b \log_f n$  (3). Although there are structures that have lower delay, they also have more connections through which it is difficult to stop a stuck-on-one fault.

### 2.2 Broadcast Domains

The tree links consist of a data link say,  $L[0\sim 7]$ , an address link  $A[0\sim 15]$ , a control link, say,  $K[0,1]$ , and priority/carry lookahead circuitry, to be used for normal operation. (Four more links will be introduced later.) Links  $\underline{A}$ ,  $\underline{L}$  and  $\underline{K}$  are bidirectional amplifiers which are independently opened or closed electronically (4). A subgraph of the tree in which  $\underline{L}$  is connected is called a data domain, for which  $\underline{A}$  is connected, an address domain, and for which  $\underline{K}$  is connected, a control domain. These three domains are broadcast domains. During an event of the process, one or more cells will broadcast into a data domain in  $\underline{L}$ , the data will be wire-OR'ed, and, transferred to all cells in the data domain in the same event. The control domain in  $\underline{K}$  and

address domain in A work the same way.

### 3. THE CELL

#### 3.1 Construction

The cells consist of a microcomputer CPU and some random access memory (see Figure 1). For simplicity, we will assume that the random access memory is, say, a 1K x 8 bit page with a 16 bit address. The high order 6 bits of the address of the words on this page will be the same, and will be called the page number. The page number will be stored in a register PAGE[0~5] in the memory decoder. When an address A[0~15] is presented in an address domain, the memory with PAGE[0~5] equal to address A[0~5] will read, or write data from the link L into a word on that page chosen by bits A[6~15].

The microcomputer will have an arithmetic-logic unit, a microprogram store, an instruction register I[0~7], and a suitable collection of registers for programming. For simplicity, we will assume that these are a temp register TEMP[0~7], an accumulator ACC[0~7], a program counter PC[0~15] and one index register X[0~15]. Even though more registers will be required in a practical machine, these registers will be suitable to demonstrate the technique of varistructure.

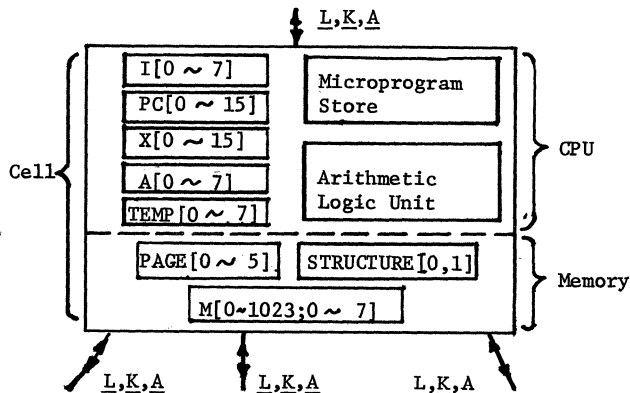


Figure 1. A Cell

#### 3.2 General Operation

Operation of the microcomputer will be similar to that of the von Neumann computer. An instruction will consist of a sequence of microinstruction cycles including a fetch cycle and a memorize cycle, recall cycle, or execute cycle. For example, a typical ADD instruction would consist of: 1) a fetch cycle, where PC is sent out as an address on link A, the returning data on L being stored in the instruction register I and decoded; 2) a recall cycle in which the word pointed to by index register X is put in TEMP and; 3) an execute cycle, in which TEMP and ACC are added, the result being left in ACC. The control link K is used to set up broadcast domains for each cycle. For a fetch cycle, K is 00, for an execute cycle, 01, for a memorize cycle, 10, and for a recall cycle, 11.

#### 3.3 Operation of STRUCTURE

Finally, each cell will have a structure state STRUCTURE [0,1]. During each cycle, STRUCTURE will determine the limit to which the address and data domains extend and the behavior of the CPU. (The

memory behaves the same for all states.) We will discuss the four values of this variable as we consider the operation of the processor. The programmer determines his configuration by loading the value of STRUCTURE in each cell before the program and data are loaded in. In general, when STRUCTURE is 00, the CPU in the cell will become "passive" and the K, A and L links above (towards the root from) this cell closed switches; when STRUCTURE is 01, the CPU will be a "byte slice". It will behave as a one byte slice of a parallel ALU other than the left slice (containing the sign bit), the microinstruction decoder will be activated, and L will be open while K and A are closed above the cell. When STRUCTURE is 10, the CPU will be a "left terminal". It will behave as the leftmost one byte slice of a parallel ALU. Only one CPU of a group will determine branching and so on. The left slice CPU will send out control lines to memory. The microinstruction decoder will be activated, to decode an instruction, and L, K and A are open above this cell. Left terminal and byte slice CPU's will be called "active" CPU's. (Another mode can be added to handle vectors, but we will not consider this simple extension.) We will discuss the operation of the recall/memorize cycle in the next section, which will show how data can be brought into the ALU in such a way that it can be treated as variable width data. We show how this is done in the following section, when the execute cycle operation is shown.

### 4. MEMORIZE AND RECALL CYCLE OPERATION

#### 4.1 Variable Structured Data

We will now consider a technique whereby the programmer can select the width and height of his random access memory. We will consider a uniform tree with fanout 3, although other configurations are obviously possible. With this configuration, the programmer can select heights of 1, 4, 13 or any number  $\sum_{i=0}^n 3^i K$  words, and widths of 1, 3, 9 or any number  $3^n$  bytes per word. Of course, with a fanout of  $f$ , the height can be  $\sum_{i=0}^n f^i K$  and the width can be  $f^n$ .

#### 4.2 Selection of Memory Height

Starting at the leaves of a tree, all cells can have STRUCTURE equal to 00 so that the CPU will be passive and the memory will be made available to the data and address links. Suppose there is one that is  $n$  levels (say 2) away from the leaves that has STRUCTURE not equal to 00, all those below it having STRUCTURE equal to 00 (see Figure 2).

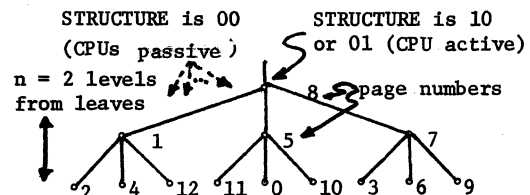


Figure 2. Height selection of byte slice



The A, L and K links between the cells in this subtree are connected together, and only the topmost CPU is active, the others being passive. Since, in the recall cycle, any cell can read a word into the link, and from there to the active CPU, we can assign different page numbers to different cells. They do not need to be in any order (see Figure 2). The numbers next to the nodes are pages numbers. On the recall cycle, an address is sent by the active CPU to all cells on the connected A link in the address domain, and one of them will match the high order bits of the address with its page number PAGE. It will send a word on the connected L link in the data domain. The active CPU will load this data into its TEMP register. The memorize cycle will, of course, be similar.

#### 4.3 Selection of Memory Width

A collection of subtrees that constitute a data domain can be in a large subtree that constitutes an address domain.

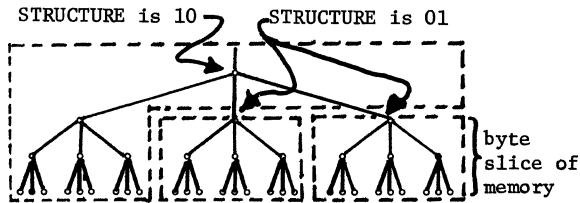


Figure 3. Width selection combining byte slices

This can be accomplished by making the root cell of the larger subtree have STRUCTURE 10 making it a left slice CPU. It will also completely delimit the link communication above it. In this way, when a cell with STRUCTURE 10 broadcasts an address on the A link below it, it goes to all data domain subtrees simultaneously, and each recalls or memorizes a word of data separately at the same logical address.

It should be noted that the leftmost data domain subtree is larger than the other two data domain subtrees in Figure 3. All that we require is that each data domain has a unique page number for every page of data that contains data to be used with a memorize or recall cycle. The leftmost data domain will have an extra memory page which, at this point, need not be assigned. Indeed, because of the failure of some cells, it may well be that each data domain has a different number of nodes. If  $n$  is the minimum of the number of good nodes in any data domain, then pages in all data domains can be numbered from 0 to  $n-1$ .

It should be evident that during a recall or memorize cycle, the size of memory can be any of a number of widths and heights. This selection is made by assigning the values of the value of STRUCTURE in each cell. Finally, since the topmost cell in an address domain disconnects all links above it, it should be evident that a large tree can be space-shared, where different problems are run in different address domain subtrees.

#### 5. EXECUTE CYCLE OPERATION

In the execute cycle, we assume that an operand is available in ACC and possibly one is available in TEMP. For logic operations such as negate or AND, the active CPU's will simply compute the result in parallel. (We will discuss in a later section how instructions appear

in all active cells so that the cell CPU can execute the correct operation.) The only real problem is the communication on the carry link or the right shift link between active CPU's. We will consider the carry link. The right shift is similar. To see how this operation is done, we first look at the operation of the carry lookahead adder. In particular, we choose a standard carry lookahead module (74182). (We assume the reader is familiar with the operation of carry propagates, generates and carry inputs of this module.)

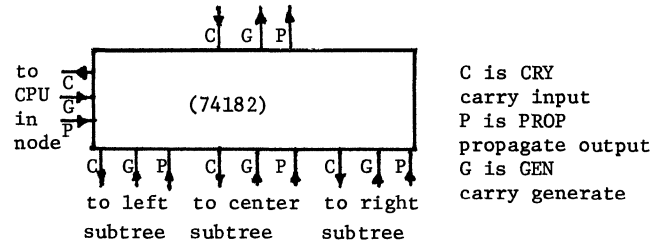
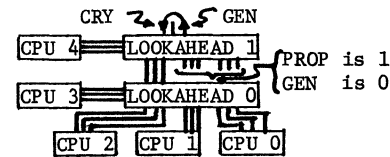
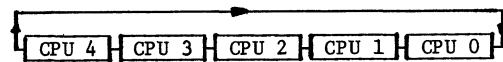


Figure 4. Connections of a carry lookahead unit

The carry-in, generate and propagate links of the CPU in that node are connected to the fourth (leftmost) set of links to this module, and the carry-in, generate and propagate links going rootward from each subtree of the node are connected to the carry lookahead module of that node. The set of links, group carry-in, group generate, and group propagate, are connected to the next rootward node carry lookahead module. The effect of this connection is to put all nodes in so-called left list matrix order (5) so that they can be thought of as being ordered in a chain, even though the delay is logarithmically related to the number of cells.



a) A subtree



b) Equivalent chain

Figure 5. Carry logic

The manner in which the tree is made to look like a chain of CPU's is as follows (see Figure 5). LOOKAHEAD 0 distributes carry signals to its attached CPU's and forms a group generate and group propagate shown above it. This is input to LOOKAHEAD 1 as though it were a CPU input in LOOKAHEAD 0. Consequently, CPU 4 will see a carry generated by a CPU to its right if the intermediate propagates are all ones. This is equivalent to the operation of the ripple adder connected as in Figure 5b. Note also that if the group generate of LOOKAHEAD 1 is connected to its own carry input, and the unconnected inputs have Propagate = 1, Generate = 0, then the carry out of CPU 4 is the carry into CPU 0, which is equivalent to the end-around carry shown in Figure 5b. While end-around carry is not used as in one's complement arithmetic, this path enables the root cell of an address domain to set the carry input of the least significant CPU.

To operate the execute cycle, then, we use the

following scheme. The general scheme is to generate a carry input for the end-around carry in the topmost cell of a data domain, which serves as the leftmost CPU in the chain. The carry is passed from one active CPU (having STRUCTURE 01) to another, bypassing inactive (STRUCTURE 00) CPU's and subtrees (roots having STRUCTURE 10) that are disconnected from it. Bypassing is done by setting generate to 0 and propagate to 1. The rules for connecting the cells are shown in Figure 6. Figures 6a and 6d show how root cells of data domains should behave. The end-around carry is implemented for the cell itself, and the links above it are connected so that the tree above it bypasses it. This enables the root cell to set the carry into the whole adder to zero for addition, or to one for subtraction and so on. (Were this not possible, the cell in the tree corresponding to the rightmost cell of the carry chain of Figure 5b would also have to be specially designated by having a different value of STRUCTURE.)

## 6. FETCH CYCLE

The fetch cycle is responsible for getting the instruction to all active CPU's. We will show how an eight bit instruction can be sent to all of them. It is accomplished by the following scheme. We will assume that conditional branching will be done only on the sign bit for simplicity. Since the sign bit is in a left end CPU, that CPU is the only one that can evaluate a conditional branch. So it alone will send out the address of the instruction in a fetch cycle, as it did for the recall/memorize cycles. The address is derived from PC and PC is incremented if no branch is taken, or from X if a branch is taken, as in standard computer organizations. However, the data domain is made the same as the address domain so that the instruction is sent to all cells. This is done by delimiting the data link I only above a cell with STRUCTURE 10. All cells will load the instruction into their I register.

In order to support this scheme, it is necessary that the words addressed as instructions are in only one memory page in the entire address domain. This is accomplished by storing instructions in pages that are not used for data. For example, in the tree in Figure 3, each byte slice subtree may have a page address from 0 to 11 for data. Each byte slice subtree then has one page left over, and the leftmost byte slice has two pages left over. Thus, four pages are available to store instructions (assuming no faulty pages are in the tree). These pages will be given page addresses 12 to 15. All instructions will then have to be in pages 12 to 15 so that just one 8 bit word will be read during a fetch cycle.

It should be evident that at the end of a fetch cycle, all active CPU's have the same instruction. Thus, they can decode the instruction in parallel. The decoded signals will control the CPU's. Only the left slice CPU will control the address and memory, however.

## 7. SET-UP OF THE STRUCTURE

The normal operation of the machine is defined in terms of address, control and data domains, as we have shown in the earlier sections. These are established by setting STRUCTURE and PAGE in each cell. This can be done to avoid faulty cells. We consider how this is done now. The problem of identifying faulty cells and excising them is first discussed as part of the set-up procedure. Then the problem of arranging the structure is considered. Finally we consider the input/output strategy, which is also used to request a set-up.

### 7.1 Identification of Faulty Cells

As we noted earlier, the bidirectional amplifier (4) can be used to isolate faulty cells, especially those in which an output amplifier is stuck-on-one. The bidirectional amplifiers in the links can be arranged so that they transmit information only from the rootward cell to the leafward direction. Hence, a stuck-on-one fault in a tree structure cell tends to be in a leaf cell, so that most of the machine is able to receive information from the root of the entire tree. A fault diagnosis program can be entered at the root to exercise all the memory chips independently, as memory is commonly exercised in a diagnostic program in any computer. The CPU can also be exercised, either with instructions from the root of the entire tree or from routines stored in the memory pages when they have been checked out. A faulty memory can be given a PAGE number that is not used by the programs to be executed, so that it is not used. A faulty CPU can excise itself

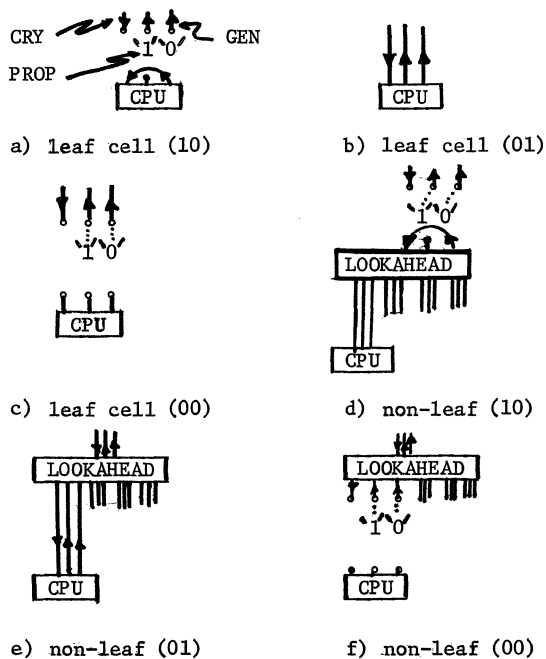


FIGURE 6. Connection of leaf and non-leaf cells for different values of STRUCTURE (given in parentheses) during the execute cycle.

Secondly, a data domain root cell, with STRUCTURE 01, will connect as in Figure 6b or 6e. Because it is a CPU in a byte slice of a larger chain, it contributes carry generate and propagate signals, and accepts carry-in signals in the normal way.

Thirdly, an inactive cell, with STRUCTURE 00 will behave as in Figures 6c and 6f.

It should be apparent that the carry chain is connected for addition, subtraction, and shift left (i.e., add a number to itself). The right shift requires a link that is easily connected in the reverse direction to the carry link.

by setting STRUCTURE to 00. A subtree containing a faulty link (stuck-on-one) can be excised by setting STRUCTURE to 10 in the root of the subtree. The test for a faulty link is simple. If a cell and each of its sons in the tree structure are found to be faulty, the cell will have its STRUCTURE set to 10.

This scheme simplifies the fault detection program to the problem of checking just one cell. All cells are then checked in parallel in this machine.

### 7.2 Setting of STRUCTURE and PAGE

Several techniques are possible for setting these values in each cell. A simple technique is serial transmission of the address and data on two broadcast links, U and V, and a propagating (store and forward) link T (see Figure 7). We discuss a binary tree here for simplicity. This scheme is similar to Berkling's address scheme (6). U contains (sequentially) the address and values of STRUCTURE and PAGE. V is 1 if the corresponding bit is an address, zero otherwise. At the beginning, V is 1, T is 1 into the root cell. The bit on U carries T to become 1 on the link to the left son of the root if U is 0, and the right son of the root if U is 1. The sequence of address bits can be sent on U as V is 1, while T walks down the tree. For example, to address cell A in Figure 7, U would be the sequence 1, followed by 0. When the desired cell has been reached, V is set to zero. The register pair STRUCTURE, PAGE then would operate as a shift register, shifting the value of V into it. The cell would be rendered inoperative until the eight bits have been shifted in. This process would be repeated, first setting T to 1 and V to 1 at the input to the root for supplying addresses and values, to all cells.

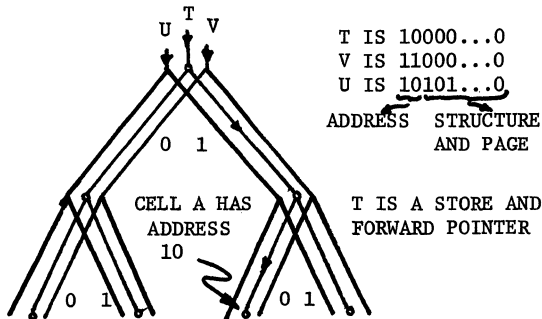


Figure 7. Selection of Cells by Tree Address

The above procedure can be executed to set up a structure. It can even be set up in one subtree while other subtrees are doing useful work. It only requires that a way is available to move data into the pages set up, and conversely, for a processing tree to request the root cell to change the structure. This is part of the general input/output problem, which we consider next.

### 7.3 Input and Output

One feasible scheme for input and output would be to build a tree of bidirectional links (I/O link) in parallel to the tree used for processing. The input scheme would be to select a cell, as we did in the last section, and broadcast the data on the I/O link. Only the selected cell will respond. For output, the entire tree can have a hardware priority structure such that a cell can request a path from the root to select the cell. Then it can broadcast into the I/O link tree. Only the root of the tree will respond. Data can be output, and requests to change the structure can be

sent to the root cell in this manner. However, a large number of possibilities exist, with different advantages in terms of cost or throughput. It is necessary, however, that faulty cells can be deleted from this tree as well as the tree used for processing to avoid stuck-on-one faults in the link. The same procedure that we discussed for checking for stuck-on-one faults in the processing tree can be used in this particular I/O link tree.

### 8. CONCLUSIONS

We have described a novel computer architecture that offers most of the advantages of cellular machines, yet is sufficiently similar to a standard von Neumann computer once it is set up that it will be possible to program it. The techniques described in this paper show it is possible to select the height and width of memory to be used in a task. It is also possible to avoid faulty cells in this structure.

### REFERENCES

1. LIPOVSKI, G.J., "The Architecture of a Large Associative Processor," SJCC, 1970, Vol. 37, pp. 385-395.
2. CHEN, T.C., "Distributed Intelligence for User-oriented Computing," FJCC, 1972, Vol. 41, pp. 1049-1056.
3. LIPOVSKI, G.J., "The Architecture of a Large Distributed Logic Associative Processor," Coordinated Science Laboratory Report R-429, July, 1969.
4. VICE, W.E., LIPOVSKI, G.J. and BRODERSEN, A.J., "On Integrated Circuit Bidirectional Amplifiers," IEEE Journal of Solid State Circuits, Vol. SC-8, No. 5, Oct. 73, pp. 381-388.
5. IVERSON, K.E., A Programming Language, Wiley, 1962.
6. BERKLING, K.J., "A Computing Machine Based on Tree Structures," IEEE Trans. on Computers, Vol. C-20, No. 4, pp. 404-418, April 1971.



# A HARDWARE LABORATORY FOR COMPUTER ARCHITECTURE RESEARCH

Jean Vaucher  
Christian Rey  
*Universite de Montreal  
Montreal, Canada*

## ABSTRACT

Because of dramatic reductions in cost of mini-computers, peripherals and logic modules, it is becoming evident that many problems confronting the computer system designer will be solved in the future by hybrid designs involving not only software but also specialized computers with architectures best suited to each application. Accordingly, hardware research must no longer be considered as a separate discipline by system programmers but as a tool in exactly the same way as languages. To illustrate this philosophy, a hardware laboratory has been set up at the University of Montreal. The primary interest of the founders was in designing and building small specialized computing systems.

This paper describes some of the aspects of the laboratory with emphasis on two major developments:

- (1) The design of a programmable I/O switch between two mini-computers.
- (2) The addition and monitoring of a writable control store connected to one system.

## 1. INTRODUCTION

Because of dramatic reductions in cost of mini-computers, peripherals and logic modules, it is becoming evident that many problems confronting the computer systems designer will be solved in future by hybrid designs involving not only software but also specialized computers with architectures best suited to each application. It follows that hardware research is important to any Computer Science Department as a whole. However, research in computer architecture is not possible on existing Computer Centre machines. These are committed to providing a service to the software community and cannot allow internal modifications or the attachment of non-standard peripherals. Further, because of the time required to produce and market computers, these machines are obsolete and their design is 5 to 10 years behind that of the prototypes being developed by the manufacturers. By working only with available machines, a university researcher is, in a sense, developing algorithms for yesterday's computers.

To fill this hardware gap, at the end of 1971, a hardware laboratory was set up by the authors and Prof. Paul Bratley at the Computer Science Department of the

University of Montreal. The primary interest of the founders was in designing and building small specialized computing systems but the aims of the laboratory went beyond this in that the laboratory should provide hardware facilities to aid research in all areas of the department in roughly the same way as a computer center provides software facilities. In short, we wanted to extend the range of options available to our researchers to the fields of hardware and firmware [14]. To determine the equipment needed in the laboratory, it was necessary to consider the probable uses that would be made of this equipment. Here is a list of research areas that were thought to be likely users of the laboratory facilities (asterisks indicate that projects are currently under way in a given area):

- (1) Microprogramming [2, 5, 12, 15]
  - Testing new instruction sets \*
  - Emulation of high level machines \*
- (2) Storage management [1, 3]
  - Virtual memory and segmentation \*
  - Hardwired garbage collection
- (3) Real-time operation [6, 7]
  - Process control
  - Time sharing supervisors \*
  - On-line processing of sounds and images \*
  - Telecommunication networks
- (4) Performance measurement [8]
  - Hardware monitors \*
- (5) Trial designs with new technology
  - LSI, COSMOS, MOSFET \*
- (6) Specialized peripherals
  - Adaptive learning networks [9]
  - Stochastic computer \* [4]
  - Associative memory for network algorithms [11]
  - Sort-merge processor \*

To provide hardware support for the projects listed above, the laboratory should have a very flexible computer "test-bench" in addition to the traditional electronic test instruments. There are four major requirements for this "test-bench":

- it should support microprogramming;
- connection to peripherals and other computers should be simple;
- it should have software writing tools from the start (compilers, assemblers and secondary storage);
- the cost should be as low as possible.

In the rest of this paper, we describe the computing

hardware purchased for the laboratory and consider in detail two important hardware modifications that were necessary to give the system the required flexibility: the addition of a writable memory for microprograms and a modification to the I/O structure to allow sharing of peripherals between two computers.

## 2. LABORATORY COMPUTING EQUIPMENT

The "test-bench" is based on two nearly identical INTERDATA 4 mini-computers. Experience in the laboratory has shown that it is essential to have two computers so that software development can proceed on one while the other is laid up for hardware modifications. The INTERDATA 4 was chosen because it has several features useful for our applications [16]:

2.1. Microprogram control: The INTERDATA 4 runs under the control of a microprogram in a read-only memory. The machine structure permits the replacement of this memory by a writable control store, allowing dynamic microprogramming.

2.2. 16 General purpose registers: When emulating virtual machines, it is possible to reserve a few registers for special use and still leave enough for general purpose use by the programmer.

2.3. Instruction set: The instruction set is closely related to the IBM/360 and the INTERDATA is programmed like bigger machines. Out of 256 instructions possible with the 8 bit OP-code, only 86 are implemented by INTERDATA. There is therefore room for new microprogrammed instructions.

2.4. Simple I/O BUS [17]: The standard multiplexer I/O BUS is simple and easy to connect to. It has only 27 lines: 8 for input, 8 for output and 11 for control, test and initialisation. This compares favourably with another machine which was also considered for the laboratory: the DEC PDP/11. The DEC UNIBUS has 56 lines which combined with 64 grounds make a 120 wire BUS. The INTERDATA BUS uses the "handshaking" principle to reduce the effect of transmission delays. Eight bits are used to address peripherals so that 255 units could be connected to the BUS. A wide variety of peripheral devices are connected to the two computers. These include standard units such as: a high speed paper tape reader/punch, a matrix printer (165 char/sec), 2 disc drives with a capacity of  $1.5 \times 10^6$  bytes each and 2 selector channels to control high speed data transfers. Other peripherals are available for real time and time-sharing operation: telephone line controller, A/D and D/A converters, memory protect which is controlled as an I/O device and a high precision programmable clock.

### 3.A NEW CONTROL STRUCTURE FOR DYNAMIC MICROPROGRAMMING

The standard INTERDATA has its control microprogram stored in a magnetic read-only memory, MMF, wired at the factory. To allow dynamic microprogramming experiments, a writable microprogram memory, MMA, was added to the system. This section first describes how instruction fetch and execute is carried out by the INTERDATA; this will show the extent to which machine behaviour can be modified by microprogramming. Then the strategy presently employed to make simultaneous use of both MMF and the new MMA will be given [18,19].

3.1. Instruction processing: The INTERDATA is not a "pure" microprogrammed computer. The microinstructions are short (16 bits) and highly coded. Each microins-

truction has a 4 bit op-code which determines, along with internal status registers, the meaning of the other bits in the instruction. A microprogram location counter determines the next microinstruction cycle to be executed. The normal FETCH-DECODE-EXECUTE cycle of the INTERDATA is shown in Figure 1. Some frequently used functions of this cycle, steps 2, 4 and 6, have been speeded up through the use of special hardware. These functions which are used by the DECODE microinstruction are mainly concerned with the interpretation of the op-code part of user "macroinstructions". Naturally, they make certain implicit assumptions about instruction format and the structure of the op-code. For example, in step 2, the DECODE microinstruction determines the instruction type and causes a branch to the appropriate microroutine to fetch the operands. Later, in step 4, use is made of a special high-speed decoding read-only memory or DROM. When provided with an op-code, the DROM returns the address of the corresponding

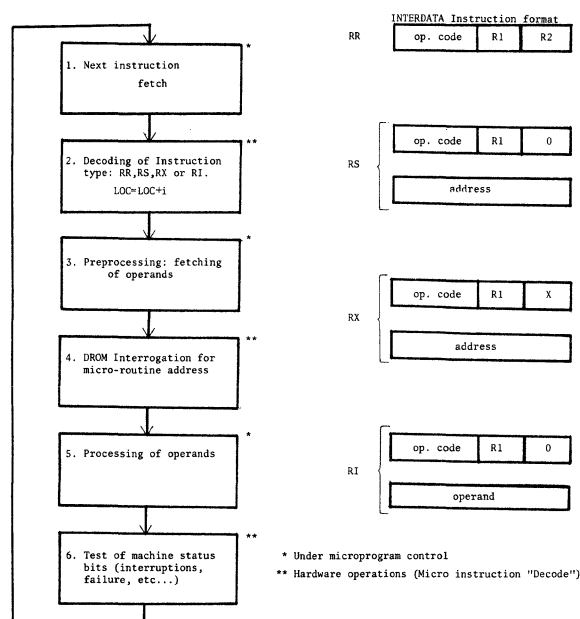


Figure 1: Instruction execution and formats

EXECUTE microroutine. Any op-code which is not implemented in the standard INTERDATA results in a branch to an "Illegal Instruction" microroutine which in turn causes an interruption in the user programme. The microprogrammer is therefore faced with a trade-off between speed and flexibility. If a standard instruction is replaced by a new one with the same format but different meaning, he can use the DECODE hardware to get fast execution. On the other hand, he can microprogram without using DECODE and obtain complete freedom in the interpretation of macroinstructions. In this case, the DECODE phase of the instruction processing will be lengthened. However, if the new instructions are fairly sophisticated and have long EXECUTE times, the penalty incurred in bypassing the DECODE hardware will be minimized. In the near future, we plan to replace the DROM by a writable decoding memory to regain some speed while retaining full flexibility.

3.1. The writable control memory (MMA): The writable memory is a thin film memory with non-destructive read-out obtained from Memory Systems Inc. of California. It has 1024 16-bit words with 400 n sec access time. It is slightly slower and smaller than MMF which contains

2K words with 300 n sec access. The speed difference does not affect proper operation of the machine and the size difference is not critical since the standard microprograms occupy only 35% of MMF. The MMA is connected to the INTERDATA in two different ways: (a) It is connected through a controller to the Multiplexer I/O BUS and can be used as a fast external memory accessed by the standard I/O commands - this is the way in which user microprograms are introduced into MMA; (b) It is also connected to the internal registers of the machine and works in parallel with MMF. When the INTERDATA requires a new microinstruction, both MMA and MMF respond and try to place their output into the internal data register. Which memory word is selected is decided by an electronic switch which is controlled by MMA's I/O controller. Initially, MMF is connected but control can be passed to MMA by a special output command to MMA. This system is very flexible. The INTERDATA can run under exclusive control of either memory or control can be passed from one to the other during execution.

3.3. Changing the instruction set: Here two possibilities must be considered: (a) The replacement of all the old instructions by a completely new instruction set and (b) The addition of a few new instructions to the already existing set.

In the first case, the complete new microprogram is loaded into MMA through the I/O BUS under control of MMF. Control is then passed to MMA by an output command.

The second case is more complex since, ideally, the existing firmware in MMF should be used for the old instructions and control should be given to MMA only for the new instructions. In this way the limited space of MMA is not wasted by duplicating the firmware already in MMF. This mode of operation gives rise to two problems: a) deciding whether an instruction is "old", "new" or "undefined" and b) finding the address of the microroutine in MMA corresponding to a "new" instruction. In MMF, this second problem is resolved through the use of the DROM. In our system, the first problem is partially solved through the use of the "illegal instruction" microroutine already in MMF. When an undefined instruction is encountered, the standard firmware causes an interrupt and stops executing the current program to branch to a monitor routine located at a predetermined address in core. This process is accomplished efficiently through the exchange of Program Status Words (PSW's). The action of the monitor routine depends on the operating system; in our BOSS supervisor, it prints an error message and aborts the user job. "New" instructions are treated by MMF as illegal instructions and it is a simple matter to modify the monitor routine so that it transfers control to MMA before printing the error message. To decide whether the instruction is "new" or illegal, the MMA microprogram uses a table with an entry for each possible op-code. In the case of a "new" instruction, the entry contains the address of the required microroutine; for an undefined instruction, it contains a special indicator. In the first case, the proper microroutine is executed and the program location counter is set to point to the next instruction in the user program. Then MMA issues an output command to return control to MMF. In the second case, control is returned to MMF immediately without altering the location counter so that the monitor routine can resume execution. This simultaneous use of MMA and MMF is summarized in Figure 2.

3.4. Conclusions: As indicated by Rosin et al. [13], the choice of an inexpensive host machine for dynamic microprogramming is severely limited. We have shown that with a few relatively inexpensive modifications, the INTERDATA 4 can be made suitable for microprogramming experiments. The resulting system does not have

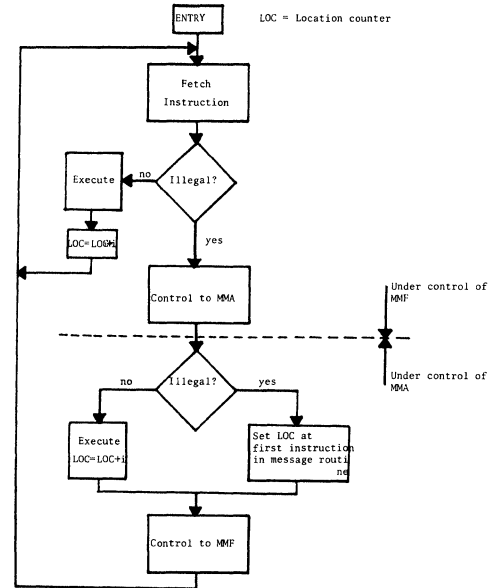


Figure 2: Simultaneous use of MMA and MMF

the full flexibility of a machine designed specifically for the purpose and the main deficiencies are:

- (a) The amount of parallel internal operation is limited due to the highly coded format of the microinstructions;
- (b) There is no support for subroutine linkage at the microprogram level;
- (c) Addressing is restricted by the division of the micromemory into 256 word blocks;
- (d) The instruction decoding mechanism (DROM) is not alterable dynamically. This, however, will be modified shortly.

In spite of these deficiencies, the present system is adequate for most of our needs: virtual memory management is being implemented through microprogramming, a new instruction set for LISP programs is in the design stage and fast instructions for waveform synthesis have been microprogrammed.

#### 4. INPUT/OUTPUT MODIFICATIONS

Although the laboratory has two mini-computers, each with its own teletype, the other I/O units have not been duplicated: for example, there is only one high speed paper tape reader-punch and only one printer. The present I/O capacity is sufficient for both machines but it is often necessary to transfer units from one computer to the other. To simplify this procedure, a programmable I/O switch (PIOS) has been designed allowing the units to be shared between the two systems. This PIOS is relatively simple and its low cost (under \$500) is compatible with the rest of the system.

4.1. INTERDATA I/O channels [16, 17]: As shown in

Figure 3, the INTERDATA does I/O in either of two ways:

(a) Directly via the multiplexer Bus (MB), misleadingly called channel by INTERDATA. The INTERDATA instruction set includes several instructions to transfer information along this BUS. Although up to 64 K bytes can be transferred with one instruction, it is important to note that the CPU is fully occupied by the transfer and cannot execute any other instructions in parallel.

(b) Through an optional Selector channel. This device is controlled via MB. It has a direct access to the memory and can transfer data between memory and units on the selector BUS (SB). After a transfer has been initiated, the channel works in parallel with the CPU. However, the parallelism is limited in that the channel does not fetch Channel Command Words from memory and each transfer must be initiated by the CPU.

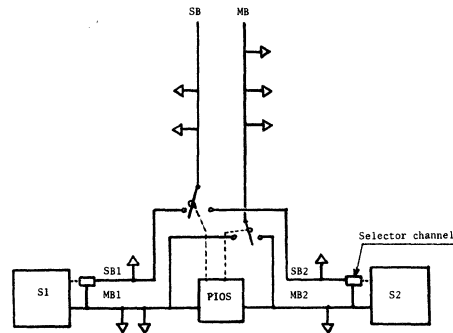


Figure 4: PIOS - General Structure

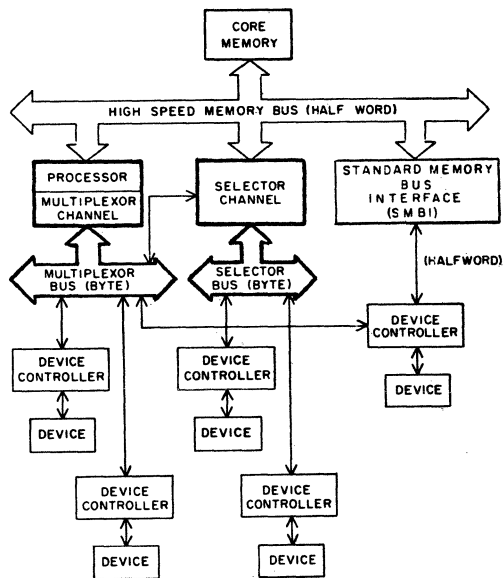


Figure 3: Systems Interface - Block Diagram

The INTERDATA also has an interrupt mechanism similar to the IBM/360's which can be used to eliminate "busy waiting" on the part of the CPU.

4.2. The programmable I/O switch (PIOS): In our system, shown in Figure 4, each bus has been divided into three sections. Each computer has exclusive use of one section and the third can be connected to either system under control of PIOS. It is on the third segments, SB and MB, that the shared units are connected. SB and MB are controlled separately. Manual switches on PIOS can force connection to either system or leave the connection to program control. It should be noted that switching applies to the buses and not to individual I/O units.

Requests for the use of the shared units are sent to PIOS via OUTPUT COMMAND and WRITE instructions along MB1 or MB2. The result of a request is indicated by the status bits of PIOS which can be tested from either system through SENSE STATUS instructions. Correct sharing of the common units requires cooperation between the two systems. To simplify the process, a

hardware "reservation" procedure has been built into PIOS. This hardware does not make the system "idiot-proof" and ways in which the system can be misused will be shown later. The reservation mechanism, however, makes it quite easy for the supervisor routines handling I/O, to avoid deadlock and excessive lock-out. This is in keeping with the general philosophy of the laboratory to integrate hardware and software design. Hardware verification of possible errors would more than double the PIOS hardware with no increase in overall efficiency. To implement the reservation mechanism PIOS maintains two registers, one for each system, where each bit corresponds to a unit on the common buses. These registers are consulted whenever a request is made to PIOS. PIOS accepts two types of request which operate as follows:

- (a) Reservation request - Here a system provides one word of information indicating the unit or units it wishes to reserve. The word has the same format as the internal reservation registers. This word is compared with the reservations made by the other system and if conflicts arise, the request is rejected; otherwise, the request is accepted and the first system's reservation register is updated;
- (b) Connection request - Although connection is made to the bus and not to individual units, this request must indicate the unit it wants to access. This is done in the same way as a reservation by providing one word with the appropriate bit "ON". Only if the unit has previously been reserved and the bus is free, is the connection made.

In this system, a bus, once connected to one of the computers, remains connected until the computer releases it. There is also no check made by PIOS upon the use the computer makes of the bus. To cancel reservations or to disconnect a bus, a request is sent with a null information word. If the common units are to be shared properly, a certain protocol must be observed by both systems. This protocol is best explained with an example. Assume that computer A wishes to use the printer connected to the shared multiplexer bus. At the same time, computer B is using the paper tape reader-punch on the same bus. The necessary operations for computer A are shown below:

1. Start of job;  
Reserve printer;

-  
-  
For each output to the printer the following 3 step sequence should be observed:

2. Connection to MB by requesting printer;
3. Output to printer;
4. Disconnect MB;

- -  
N-1. Cancel printer reservation;



N. End of job;

By reserving the printer for the duration of the job we ensure that computer B output will not be mixed with computer A's. By disconnecting the bus (step 4) after each output, we allow computer B to access the punch. The process works well only as long as the two systems cooperate. If one system neglects to disconnect the bus, the other will be locked out. Also, since PIOS does not check the activity on the bus, it is possible to bypass the reservation system by requesting connection to one unit and doing I/O to another. In step 3 of the example, computer A could have used the punch reserved by computer B. However, the required cooperation can easily be enforced by appropriate supervisor software in both systems.

The problem of deadlock can also be avoided by software. Deadlock commonly occurs when: system 1 has X and wants Y, and, system 2 has Y and wants X. The solution makes use of the reservation procedure and proceeds in two steps: (1) cancel all reservations, and, (2) reserve all units required with a single request.

4.3. Conclusions: A programmable I/O switch has been designed and built to allow two mini-computers to share I/O units. The I/O switch makes use of a novel hardware "reservation" scheme to facilitate cooperation between the two computers. The complexity (120 integrated circuits) and cost ( $\approx$  \$400) of the I/O switch are low, in keeping with the rest of the system. The "reservation" hardware is not meant to prevent I/O programming errors; but it does make it easy to write supervisor software to ensure correct operation. The I/O switch is modular in design and it is planned to add more features to it: in particular, a channel to channel communication facility to enable the two computers to exchange data efficiently. Research is also being done to find a foolproof scheme to handle interruptions from the shared units and direct them to the proper system.

## 5. SUMMARY

In spite of its short existence, the hardware laboratory has proved to be a very useful research facility. The flexible computing "test-bench" described in this paper is already being employed by several projects. One project in the field of graphics is of special interest since it uses most of the system's facilities: it involves the design of a hardware vector generator driven by software and firmware routines. Two major software projects are under way: WADOCH, a real-time operating system and a compiler for the PASCAL language. To run efficiently, these programs require more memory than presently available on our machines. This should be solved by the next major improvement to the computing system: the addition of virtual memory. At first, the address modification will be done entirely by microprogram but later associative registers will be added to speed up the process.

The current projects are not only from the systems section of the department. Two numerical analysts are studying the possibility of microprogramming special instructions for interval arithmetic [10] and a group in artificial intelligence is building an adaptive learning network to be controlled by the INTERDATA. The laboratory has also been invaluable for a course in Computer Architecture given to undergraduate students.

By basing our computing test-bench on standard mini-computers and peripherals, the investment required to set up the laboratory was modest: \$110,000 divided into \$85,000 for the computing hardware and \$25,000 for electronic equipment and tools. The flexibility required for research was achieved through the two modifications described in this paper. The main charac-

teristic of the laboratory is that it permits integrated research into three related areas: software, hardware and firmware. This approach has already proved fruitful and should remain successful in the future.

## 6. ACKNOWLEDGEMENTS

We are grateful to the Québec Ministère de l'Éducation and the National Research Council of Canada for the grant required to set up the laboratory and support our research.

## 7. BIBLIOGRAPHY

- [1] Bell G., Newell A. "Computer Structures: readings and examples", McGraw-Hill (1971).
- [2] Boulaye G., Mermet J. (Eds.), "Microprogramming", Hermann, Paris (1972).
- [3] Dalley R., Dennis J.B., "Virtual Memory, Processes, and Sharing in Multics", Comm. of ACM, Vol. 11, Number 5, May 1968.
- [4] Gaines B., "Advances in Information Systems Science", Vol. 2, Chap. 2, Ed. J. TOU, Plenum Press, 1969.
- [5] Husson S.S., "Microprogramming: Principles and experience", Prentice Hall (1970).
- [6] Liskov B.H., "The Design of the Venus Operating System", Comm. of ACM, Vol. 15, Number 3, March 1972.
- [7] McGee W.C., and Petersen H.E., "Microprogram control for the experimental sciences", Proc. AFIPS 1965 FJCC, Vol. 27, p. 77.
- [8] Miller E.F., "Bibliography on techniques of Computer performance analysis", Computer, 5, 5, pp. 39-47, (Sept. 1972).
- [9] Minsky M., Papert S., "Perceptrons", The MIT Press (1969).
- [10] Moore R.E., "Interval analysis", Prentice Hall (1966).
- [11] Orlando V.A. and Berra P.B., "Associative Processors in the solution of Network Problems", 39th National ORSA Meeting, Dallas, Texas, May 5-11, 1971.
- [12] Rosin R.F., "Contemporary Concepts in microprogramming and emulation", Computing Surveys, Vol. 1, Number 4, Dec. 1969, pp. 197-212.
- [13] Rosin R.F., Frieder G., Eckhouse R.H., "An Environment for research in microprogramming and emulation" pp. 341-395, in Reference [2].
- [14] Taylor L. et al., "Minicomputers in the digital laboratory program", Report by the Cosine Task Force on Mini computers, Computer, pp. 28-42, Vol. 6, Number 1, January 1973.
- [15] Weber H., "A microprogrammed implementation of EULER on IBM S/360 Model 30", Comm. ACM, 10, 9, pp. 549-558, (September 1967).
- [16] -----, "INTERDATA Reference Manual", Publ. #29-004R02, Interdata Oceanport, New Jersey, USA.
- [17] -----, "Systems Interface Manual", Publ. #29-003R02, Interdata, Oceanport, New Jersey, USA.
- [18] These abbreviations come from the French: MMF = micromémoire fixe and MMA = micromémoire altérable.
- [19] Rey C. and King C., "Implantation d'un système à Microprogrammation Dynamique sur un Interdata 4", Publication #146, Dépt. Informatique, Université de Montréal, 1973.



# SIMULATION EXERCISES FOR COMPUTER ARCHITECTURE EDUCATION

P. J. Knoke  
Radiation, Inc.  
Melbourne, Florida

## ABSTRACT\*

In a case studies approach to computer architecture education, there is a need for small-scale simulation exercises to illustrate significant concepts and to provide hands-on student experience with architectural tradeoffs. Two such exercises are discussed, and one is described in some detail. The exercises cover virtual memory and multiprogramming systems' architecture, and are suitable as projects students can do within a ten-week academic quarter. Some hindsight based on student reaction to these exercises is provided, together with estimated costs to the educator and students for exercise development and execution.

## I. INTRODUCTION

A case studies approach to computer architecture education has a number of advantages, one of which is that there exists today an excellent text in support of such an approach.<sup>(1)</sup> However, a disadvantage of the approach is the difficulty of going beyond computer system overviews and comparisons to ensure a solid grasp of significant concepts and to develop a student's feel for actually doing computer architecture. One way to accomplish these ends is to supplement the student's diet of case studies with a special interest architectural topic to be explored in some depth during the term. Examples of such topics are virtual memories, multiprogramming, parallel processing, higher level language architectures, etc.; and computer simulation can be a powerful tool for in-depth exploration of these topics if simulation exercises can be defined which are consistent with the various constraints of an academic environment.

Ideally, simulation exercises for these tutorial purposes should be comprehensive, realistic, computationally feasible and student compatible.

A well-designed simulation exercise can be expected to improve a student's understanding of important architectural concepts and his feel for the impact of architectural tradeoffs in varying computer system environments. It is interesting that even the (inevitable) imperfections of the models used in the simulation exercises can serve as tutorial assets, because an imperfect model can serve as a "straw man" and provoke student thought as to what form a better model should take.

This paper is based on the author's experience with two small-scale computer simulation exercises which have been used in support of case studies type computer architecture courses taught by the author at Wright State University. The exercises cover virtual memory and multiprogramming systems' architecture, and each exercise was developed and executed as a supplementary project during a ten-week academic quarter. Because it is assumed that the primary audience for this paper is computer architecture educators, the emphasis of the paper is objectives and strategy for both exercises, and only the virtual memory exercise is described here. Additional details on both exercises are available in<sup>(2)</sup> for those interested.

## II. GENERAL OBJECTIVES, STRATEGY AND COSTS

Since the general objectives and strategy were the same and costs were similar for both exercises, it is most efficient to cover these matters in a separate section. Specific objectives and strategy will be covered later in sections dealing with the individual exercise particulars.

### General Objectives

Some of the general objectives of both exercises can be compactly described with the aid of formulas, as follows: Let

$$Y_i = \text{value of the } i\text{th system performance figure of merit} \quad (1)$$

$$X_i = \text{value of the performance or capacity of the } i\text{th system component} \quad (2)$$

$$W = \text{system workload or job environment} \quad (3)$$

$$P = \text{system price} \quad (4)$$

$$Y_i = f_i(X_1, X_2, \dots, X_n, W) \quad (5)$$

$$Y_i/P = \text{system performance/price with respect to the } i\text{th system figure of merit} \quad (6)$$

$$\frac{\partial Y_i}{\partial X_j} = \text{sensitivity of the } i\text{th system performance index to changes in performance of its } j\text{th component} \quad (7)$$

\* The work reported on here was done when the author was Visiting Associate Professor in the Computer Science Department of Wright State University, Dayton, Ohio, 9/72-6/73.

$$\frac{\partial Y_i}{\partial W} = \text{sensitivity of the } i\text{th system performance index to changes in system workload} \quad (8)$$

$$\frac{\partial Y_i}{\partial P} = \text{sensitivity of the } i\text{th system performance index to changes in system price} \quad (9)$$

Then the general objectives of both exercises can be succinctly stated as follows:

1. To improve a student's understanding of the architectural concept being explored by having him construct a model of a system which incorporates this concept; i.e., by having him write a program to implement the  $(f_i)$ .
2. To improve a student's understanding of the behavior of a system based on this concept by having him run his model with varying component values and workloads, and having him observe associated variations of performance indexes, i.e., by having him determine various values of  $\frac{\partial Y_i}{\partial X_j}$  and  $\frac{\partial Y_i}{\partial W}$ .
3. To improve a student's understanding of system performance/price and variations thereof by having him include price data explicitly in the above calculation; i.e., by having him determine values of  $\frac{Y_i}{P}$  and  $\frac{\partial Y_i}{\partial P}$ .

### General Strategy

In the context of these objectives, the general strategy took the form of the following seven steps:

#### Instructor's Role

1. Decide on a suitable special interest architectural topic for in-depth exploration during the quarter.
2. Provide a specific description of model inputs and outputs.
3. Provide a loose description of the model to be implemented, together with details of configuration, component performances, and component prices.
4. Provide background information directly relevant to the selected topic in whatever form it may be available (e.g., papers, lectures, movies, etc.).
5. Provide loose guidelines regarding the desired form and coverage of the report to be prepared describing the exercise.

#### Student's Role

6. Write simulation program implementing the model in the language of his choice (usually Fortran) to be run on the computer of his choice, and exercise the model with varying component parameters and varying workloads.
7. Prepare a report describing the simulation experiment, interpreting the results, and critiquing the experiment, including recommendations for its improvement.

In the case of both exercises, as conducted at Wright State University, the main criteria for the instructor's decision in step 1 were significance and timeliness of the topic in the light of contemporary computer architecture developments, and compatibility of the topic with the case studies to be conducted concurrently. The former criterion tended to guarantee relevance and an ample supply of current background material; thus it eased execution of step 4, while at the same time increasing student interest. In step 2, it was necessary to specify the interface between model and its inputs and outputs early and quite precisely, so that students could get on with the matter of model building as soon as possible in the quarter. The model descriptions given in step 3 were quite loose, thereby relieving the instructor of a potentially heavy burden while at the same time increasing student benefits from the exercise by requiring them to do their own analyses of fairly complex problems. The report guidelines of step 5 were necessary because of the low average level of student experience in architectural analysis, but the guidelines were kept as loose as possible to encourage independent thought about the simulation experiments themselves.

### Exercise Costs

Each exercise was presented and explained to students on a piecemeal basis rather than all at once, i.e., the various necessary descriptions were introduced by portions of lectures, handouts, etc., over the 10-week course period. The cost to the instructor of running the exercises was about 20 percent of available lecture time, plus the time required to develop and publish descriptive materials in the form of handouts (examples of the latter are given in<sup>(2)</sup>).

Other costs of running the exercises are the amount of student time required, and the amount of computer resources used. These costs, of course, vary widely, depending greatly on student's modeling and programming skills. One of the better students in the class gave the following estimates of these costs (programs in Fortran, run on a PDP-10):

	<u>Programming Time</u>	<u>Program Run Time</u>
Virtual Memory Exercise	10-12 hours	7 minutes
Multiprogramming Exercise	10-12 hours	15 minutes

However, the above figures are for two of the most sophisticated models developed, and so the run time figures are not typical of the class as a whole. The following program run time figures are more typical of student models:

	<u>Program Run Time</u>
Virtual Memory Exercise	15 minutes on IBM 1130
Multiprogramming Exercise	5 minutes on IBM 360/65

### III. VIRTUAL MEMORY SIMULATION

The virtual memory exercise was described to students in a series of partial lectures, a movie and handouts, with the latter containing configuration, performance and price data and the former serving to explain the virtual memory concept and the associated handouts. Because the IBM 1130 computer system was the only computer system with which all students in the class were reasonably familiar, and because the question of applicability of the virtual memory concept to small-scale

computer systems is an interesting one, the exercise took the form of a feasibility study for adding a virtual memory to a small IBM 1130 system.

### Specific Objectives and Strategy

Specific objectives can be obtained from the general by assigning particular meanings to the performance figures of merit, etc., given in Section II, equations 1-9. This is done next. A fixed-page-size type of virtual memory system is assumed.

- Y = number of memory accesses per second
- $X_1$  = page replacement algorithm
- $X_2$  = page size (words)
- $X_3$  = real memory size (words)
- W = address stream
- P = system price
- $Y = f(X_1, X_2, X_3, W)$
- f = mathematical model of the virtual memory system for these purposes

Essentially, in this exercise, it is assumed that an 1130 user has a choice between an 1130 system with a real memory of 64K 16-bit words, and a Virtual Memory (VM) 1130 system with a virtual memory of 64K words and a real memory of less than 64K words plus an address mapping device, called a VM box, with a certain assumed price and performance. Price and performance data used was that from IBM 1130 price lists and manuals for standard system components. All price data was reduced to equivalent dollars per month, and in cases where only an actual or estimated selling price was known, an estimated dollars-per-month figure was obtained by dividing the selling price by 40. In cases such as that of the VM box, an estimate of cost and performance was made and the selling price was obtained by assuming a 4:1 ratio of selling price to cost, this being quite typical in the computer industry. Details such as this provided an opportunity for the instructor to point out to the student the important distinction between price and cost in computer architecture studies. As a result of the above, it was estimated that a 4K VM-1130 system could reasonably rent for about \$2500/month, while a 64K non-VM-1130 could reasonably rent for about \$5300/month, with both offering the user 64K words of addressable main memory. Furthermore, it was estimated that the VM-1130 system could have additional 4K word increments of real memory for about \$205/month each. Thus, the stage was set for examining the performance and price/performance consequences of the VM-1130 with the aid of simulation.

### Model Inputs and Outputs

The particular inputs and outputs are loosely defined in the previous section. More specifics of these were described in

student handouts, with the understanding that students could deviate from these specifics if such deviations seemed desirable on the basis of preliminary experiment results. Students were advised to use the following parameter values initially in exercising the model:

- $X_1$  (page replacement algorithm): FIFO, LRU, PA\*
- $X_2$  (page size, words): 16, 32, 64, 128, 256, 512
- $X_3$  (real memory size, words): 4K, 8K, 16K, 32K
- W (address stream): AG-1, AG-2, AG-3A

Most of these inputs are self-explanatory, except that the PA page replacement algorithm is not as widely known as the other two, and the synthetic address stream (W) was invented for purposes of this exercise. The PA algorithm (3) is essentially one wherein information as to whether or not a page in main memory has been altered in the course of references made to it. If a particular page has not been altered, then when that page is ousted from main memory it is unnecessary to write it back in the disk supporting the virtual memory scheme because that disk already contains an identical copy of the page in question. Thus, the saving of this type of page status information can lead to a substantial reduction in the number of disk accesses required, and therefore increases virtual memory performance.

The virtual memory system model is driven by a synthetic address stream, which is essentially a sequence of numbers on the virtual memory address space (0,65535) obtained by suitable modifications of a random number generator. Address generators AG-1, AG-2 and AG-3A are simple empirically derived address streams which are intended to be statistically similar to various possible actual address streams. Roughly speaking, they may be characterized as a severe thrasher, a moderately severe thrasher, and a moderate thrasher, respectively. There is, of course, no implication that these labels are valid relative to real-world "typical" address streams, since it is not known to this writer what these real-world entities actually are. The generalized algorithm for all address generators is given below:

### Generalized Address Generation Algorithm

- Step 1. Define address space =  $S = (0, \text{MAX})$
- Step 2. Select starting address randomly on  $S = A_1$
- Step 3. Given address  $A_1$ , Select  $A_{1+1}$  as follows:
  - with 50 percent probability,  $A_{1+1} = A_1 + 1$
  - with 25 percent probability,  $A_{1+1} = f_1(0, A_1)$
  - with 25 percent probability,  $A_{1+1} = f_2(A_1, \text{MAX})$
- Step 4. Return to Step 3.

In the above,  $f_1$  and  $f_2$  are functions which map the address domain into itself. The various address generators differ in the particular form of the functions  $f_i$ .

\* FIFO = First In First Out, LRU = Least recently used, PA = Push Alter. In the use of PA, it is assumed that 90 percent of the time a page is not altered when referenced. See (3) for further details.

## Model Description

The model used to simulate the virtual memory system is quite simple, and is given in Figure 1. The numbers used in the bottoms of the flow chart boxes indicate the time required for the system being simulated to execute the indicated step. It is assumed that the VM box can determine whether or not the desired page is in memory in the time of  $1 \mu\text{s}$ , and that it can execute the desired page replacement algorithm and perform housekeeping (update page tables, etc.) in an additional  $1 \mu\text{s}$ . The value of the PA algorithm is clearly revealed by this model, since, in this case, block 4 can frequently be bypassed.

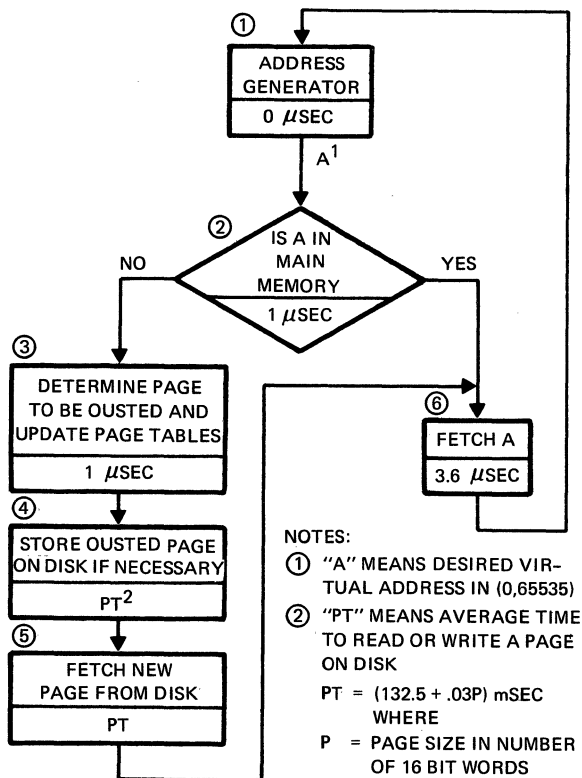


Figure 1. Virtual Memory System Model

## Results and Discussion

Some simulator outputs for varying core sizes, page sizes, page replacement algorithms and address generators (program environments) are given in Table 1.\* The virtual memory figure of merit was taken as the number of main memory accesses per second. The LRUA algorithm indicated in Table 1 was invented by the student whose results these are, and it uses a combination of the basic concepts of the LRU and the PA algorithms.

For scientific research purposes, the experiment, itself, clearly needs much more refinement. Nevertheless, these crude results are both interesting and provocative. Consider, for example, the following sample observations:

- Performance varies widely as simulation parameters are varied, from a low of about 5 to a high of about 313 memory accesses per second.

- The ATLAS algorithm is consistently one of the worst of the five algorithms examined.
- The more random the address stream, the smaller the page size should be unless the ratio of real memory size to virtual memory size is large (say,  $> 0.5$ ).
- Of all the algorithms, the ATLAS algorithm seems least sensitive to page size.

The provocative nature of the results is illustrated by the fact that students tended to do much more with the simulation than was required. For example, in Table 1, the data for LRU, FIFO, and PA algorithms was required, but the LRUA and the ATLAS algorithms were investigated on the student's own initiative.

In conclusion, the virtual memory simulation exercise accomplished three main objectives, namely: 1) complement the case studies approach to computer architecture; 2) improve student understanding of the virtual memory concept; and 3) improve student feel for tradeoffs associated with implementation of that concept.

## IV. SUMMARY AND CONCLUSIONS

A general strategy and objectives for simulation exercises useful in a computer architecture education have been described. Some costs associated with implementation of two such exercises have been discussed, and one of these exercises has been described in some detail. That exercise concerned the virtual memory concept, and it was found suitable for use as a quarter project for senior or first-year graduate level students. It was designed to complement a case studies approach to computer architecture education.

Student feedback from the exercise indicated that it resulted in an improved grasp of the associated concepts, and an increased interest in the course itself. Better initial definition of the exercises and, in some cases, better student background in both computer architecture and simulation itself would probably have resulted in increased benefits to students, but to some extent the evolutionary nature of the exercise development increased student participation in the modeling process itself, and the benefits from this tend to offset the disadvantages of the piecemeal, evolutionary approach actually used.

## BIBLIOGRAPHY

1. Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, N.Y., 1971.
2. Knoke, P., Simulation Exercises for Computer Architecture Education, Radiation, Internal Memo, 15 October 1973 (Copies available from the author at Radiation, P.O. Box 430, Melbourne, Florida 32901).
3. Belady, L. A., A Study of Replacement Algorithms for a Virtual Storage Computer, IBM Systems Journal, Vol. 5, No. 2, 1966.
4. Kilburn, T. et al., One-Level Storage System, IRE Transactions on Electronic Computers - 11, Vol. 2, pp. 223-235, April 1962 (also, Ref. (1) Chapter 12, pp. 276-290).

\* Supplied by a student, Jeffrey D. Wise, Wright State University Computer Science Department, 29 November 1972.

Table 1

## Virtual Memory Simulation-Some Results\*

## Virtual Memory Figure of Merit (Memory References per Second)

CORE SIZE	PAGE SIZE	AG-1**					AG-2					AG-3A				
		LRU	FIFO	PA	LRUA	ATLAS	LRU	FIFO	PA	LRUA	ATLAS	LRU	FIFO	PA	LRUA	ATLAS
4K	16	24	24	25	48	9	22	21	23	44	8	11	11	20	23	11
	32	24	25	25	47	9	23	22	25	45	9	12	12	22	22	12
	64	8	8	25	15	9	9	9	26	17	9	13	13	22	22	13
	128	8	8	16	14	9	9	9	19	16	9	15	15	25	25	14
	256	8	8	16	15	8	8	8	18	15	9	22	22	41	34	22
	512	8	8	16	14	8	9	9	18	15	9	-30	29	57	42	29
	1024	8	7	14	13	7	8	8	17	15	8	33	32	63	44	31
	2048	6	6	11	11	6	7	7	15	12	7	34	32	64	43	33
4096	5	5	9	8	5	6	6	11	10	6	28	28	56	35	28	
8K	16	28	28	27	56	10	26	25	26	51	9	13	13	25	27	14
	32	28	28	27	57	10	28	27	28	55	10	14	14	27	28	15
	64	29	27	27	57	10	29	26	30	58	10	14	14	25	28	16
	128	7	8	26	16	10	11	15	30	20	10	16	16	27	29	18
	256	9	9	17	16	9	10	10	21	18	11	25	25	46	40	26
	512	9	10	18	16	9	11	10	21	18	11	33	31	64	50	35
	1024	9	9	16	15	9	10	9	20	16	9	41	40	73	58	37
	2048	8	7	13	13	7	8	8	17	14	8	43	43	86	56	38
4096	5	6	12	10	5	7	7	15	11	6	54	34	67	43	34	
16K	16	44	40	36	87	16	42	40	39	84	15	17	17	31	34	17
	32	41	40	36	81	15	42	41	39	83	15	20	20	34	40	19
	64	41	40	35	83	15	44	41	38	89	15	21	21	31	43	20
	128	42	37	33	83	14	43	38	39	85	15	25	24	33	50	22
	256	13	31	34	23	13	42	34	38	85	15	33	35	55	64	32
	512	14	12	33	23	13	17	15	44	28	16	47	44	87	80	43
	1024	13	13	24	22	12	15	14	36	25	15	59	57	113	88	48
	2048	10	11	19	17	10	12	13	26	19	12	61	68	117	83	54
4096	8	8	16	13	7	10	10	24	16	10	52	44	98	61	44	
24K	16	75	72	55	150	27	157	134	107	313	55	28	28	48	55	27
	32	72	67	55	144	26	144	125	107	288	55	33	33	53	66	30
	64	74	66	55	149	26	143	109	106	286	55	30	30	46	61	30
	128	73	63	52	147	25	115	96	105	229	54	41	40	46	81	33
	256	71	58	53	143	23	127	71	102	254	56	51	50	78	102	54
	512	80	50	57	161	20	113	65	154	225	68	80	77	113	161	60
	1024	28	29	42	46	22	44	36	133	73	43	109	96	161	204	73
	2048	17	23	37	29	18	38	29	59	53	34	86	86	215	136	81
4096	20	16	32	30	20	17	22	41	25	21	89	98	163	115	70	

\* Address generators produced address streams on (0,32767)

\*\* LRU = Least Recently Used  
FIFO = First In First OutPA = Push Alter  
LRUA = Least Recently Used AlterATLAS = Replacement Algorithm  
Used for Atlas





# COMPUTER ARCHITECTURE COURSES IN ELECTRICAL ENGINEERING DEPARTMENTS

M. E. Sloan  
*Department of Electrical Engineering  
 Michigan Technological University  
 Houghton, Michigan*

## ABSTRACT

This paper traces the history of computer architecture courses in electrical engineering departments. Previously unpublished data from the Fall 1972 COSINE survey are given to show current computer architecture course offerings and texts. Computer architecture courses offered in 1972-73 are analyzed, compared with ACM and COSINE recommendations, and classified into five categories: introductory computer engineering courses with a computer architecture flavor, software-oriented computer organization courses, hardware-oriented computer organization courses, case study courses, and topical seminars. Future trends in computer architecture education are predicted.

## INTRODUCTION

Computer architecture (or computer organization) is intended in this paper to correspond to the definition given by Foster (Jan. 1973):

Computer architecture embraces the art and science of assembling logical elements into a computing device. As normally conceived of a computer architect accepts from a logical designer units such as stacks, memory blocks, and tape drives and puts them together so that they form a computer and turns this over to a systems programmer who then constructs an operating system for the machine.

Computer architecture courses comprise a major and rapidly growing division of computer engineering courses taught in electrical engineering departments. The 1972 COSINE survey of U. S. and Canadian electrical engineering departments (Sloan, Coates, and McCluskey, 1973a and b) found that the two COSINE-recommended computer architecture courses were taught at more schools than any other COSINE-recommended computer engineering courses except for introductory programming, introductory switching theory and logic, and numerical analysis. Ninety per cent of the responding schools taught both computer architecture courses, although nearly half taught one or both outside of electrical engineering.

## HISTORY OF COMPUTER ARCHITECTURE COURSES

The early history of computer architecture courses is

difficult to trace. A survey by Cook in 1963 showed that only two EE departments surveyed taught three or more computer courses and only six taught two or more in his sample of major engineering schools, granting nearly half of all ECPD-accredited bachelor's degrees. This survey and a perusal of major engineering school catalogs from the late 1950s and early 1960s suggest that few EE departments offered separate courses in computer architecture much before 1965.

Table I, adapted from the Fall 1972 COSINE survey, traces the adoption of the two COSINE-recommended computer architecture courses, called by COSINE Machine Structure and Machine Language Programming and Computer Organization; (descriptions of both courses appear in the appendix with descriptions of ACM-recommended computer architecture courses). Although COSINE had intended Machine Structure and Machine Language Programming to be prerequisite to Computer Organization (COSINE, Jan. 1970), EE departments were quicker to adopt Computer Organization and were more likely by a margin of nearly 20% to teach it rather than the software-oriented machine structure course. Machine Structure and Machine Language Programming were taught in 13% of EE departments before 1965 and are taught in 55% of EE departments today; the corresponding figures for Computer Organization are 18% before 1965 and 73% today.

TABLE I  
 ADOPTION OF COSINE-RECOMMENDED  
 COMPUTER ARCHITECTURE COURSES

	Machine Structure and Machine Language Programming	Computer Organization
Not taught	9.5%	4.9%
Taught outside EE	35.1	26.8
First taught in EE:		
Before 1965	12.8	18.3
1965-66 to 1968-69	20.9	20.7
1969-70 to 1970-71	17.6	22.0
Since 1970-71	4.1	7.3

CURRENT COURSE OFFERINGS

The remainder of the data for this paper is drawn primarily from previously unpublished data from the Fall 1972 COSINE survey. Of the 151 EE departments responding to the survey (67.4% of the 224 U. S. and Canadian departments polled), 126 departments (56.2% of the departments polled) provided varying degrees of information on their course offerings ranging from listing of titles or notation of texts to catalog descriptions and complete course outlines. These departments gave 47 different titles for courses which they identified as computer organization courses. Computer Organization, Computer Architecture, and Digital Computer Organization were the most popular titles, but misleading titles such as Programming Principles and Introduction to Information Structures and opaque titles such as Computer Engineering II were also used to designate computer architecture courses. More than one-third of the EE departments teach exactly two computer organization courses, and nearly one-tenth teach three or more.

About one-third of the departments surveyed listed the texts they intended to use in 1972-73 in their computer organization courses. Their responses are shown in Tables II and III, showing texts for first computer organization courses and for advanced computer organization courses (i. e. any courses after the first), respectively. Booth (1971), Foster (1970), and Gschwind (1967) were the most frequently reported texts for first computer organization courses; Bell and Newell (1971) was by far the most frequently reported advanced text. Some overlapping of texts can be noted with five texts being used for courses at both levels.

TABLE II  
TEXTS FOR FIRST COMPUTER ORGANIZATION COURSES

Bartee (1966)	Digital Computer Fundamentals
Beizer (1971)	The Architecture and Engineering of Digital Computer Complexes
Bell and Newell (1971)	Computer Structures
Booth (1971)	Digital Networks and Computer Systems
Chu (1962)	Digital Computer Design Fundamentals
Chu (1970)	Introduction to Computer Organization
Flores (1969)	Computer Organization
Flores (1965)	Computer Software
Foster (1970)	Computer Architecture
Gear (1969)	Computer Organization and Programming
Gschwind (1967)	Design of Digital Computers
Hellerman (1967)	Digital Computer System Principles
Lewin (1972)	Theory and Design of Digital Computers
Sobel (1970)	Introduction to Digital Computer Design
Stone (1972)	Introduction to Computer Organization and Data Structures
Ware (1963)	Digital Computer Technology and Design
Wiener (undated)	The Human Use of Human Beings
Assorted computer manuals	

TABLE III

TEXTS FOR ADVANCED COMPUTER ORGANIZATION COURSES

Bell and Newell (1971)	Computer Structures
Chu (1972)	Computer Organization and Microprogramming
Flores (1963)	The Logic of Computer Arithmetic
Foster (1970)	Computer Architecture
Gear (1969)	Computer Organization and Programming
Gschwind (1967)	Design of Digital Computers
Hellerman (1967)	Digital Computer System Principles
Husson (1970)	Microprogramming Principles and Practice
Peatman (1972)	The Design of Digital Systems
Assorted computer manuals	

RECOMMENDATIONS FOR COMPUTER ORGANIZATION COURSES

Two major national groups, the ACM Curriculum Committee on Computer Science and the COSINE Committee, have made recommendations for computer organization courses. A comparison of their courses, described in the appendix, shows much similarity and serves as a basis for considering courses actually taught in EE departments. COSINE's Machine Structure and Machine Language Programming resembles ACM's B2, Computers and Programming, while COSINE's Computer Organization corresponds roughly to ACM's 13, Computer Organization. ACM also recommends an advanced course, A2, Advanced Computer Organization.

ACM and COSINE probably never expected that departments would pattern courses exactly after their recommendations. COSINE (Jan. 1970) noted that they were more concerned with recommending topics which should be treated somewhere in the curriculum than they were with packaging topics into courses suitable for all schools. An ACM survey (Engel, 1971) of 26 doctorate-granting computer science departments showed that only 5 offered computer organization courses as specified by ACM while 17 offered similar courses.

CLASSIFICATION OF COMPUTER ORGANIZATION COURSES

The courses actually being taught in EE departments in 1972-73 differed from the recommendations in their diversity and appeared to cluster roughly into five categories:

1. introductory computer engineering courses with a computer architecture flavor;
2. software-oriented computer organization courses;
3. hardware-oriented computer organization courses;
4. case study courses; and
5. topical seminars.

Introductory computer engineering courses with a computer architecture flavor appear to be emerging rapidly

to meet a need apparently not foreseen by either ACM or COSINE. The increasing importance of digital technology has made it desirable for all EE undergraduates to have a course in digital computers beyond the usual FORTRAN or other first programming course. Survey courses, combining switching theory, machine language programming, computer organization, and sometimes other topics, are being developed, usually for sophomores and often as required courses. These courses serve to introduce computer engineering to prospective specialists as well as to overview the area for other engineering students. They can also serve as an introductory hardware course for computer science students. A suitable text is Booth (1971).

Software-oriented computer organization courses correspond roughly to COSINE's Machine Structures and Machine Language Programming and to ACM's B2. The software emphasis makes the course relatively more likely to be taught in computer science departments than the other types of computer organization courses. This course is taught most frequently at the sophomore or junior level and usually includes substantial hands-on programming experience with a minicomputer. Gear (1969) and Stone (1972) are typical texts.

Hardware-oriented computer organization courses comprise the most commonly taught group of computer organization courses. They are taught at both the introductory and advanced levels. The introductory courses correspond roughly to COSINE's and ACM's Computer Organization and are usually taught to juniors and seniors. Their approach to computer organization is more passive than the advanced courses; emphasis is placed on the student's understanding of the way a computer operates rather than on his preparation for designing computers. The courses may or may not include switching theory and logic design; (ACM recommended including logic design topics while COSINE recommended that logic design be a prerequisite to the course). The course is often accompanied by a laboratory, especially when logic design is included. Chu (1970) and Gschwind (1967) are suited for introductory hardware-oriented computer organization courses.

Advanced hardware-oriented computer organization courses tend to have a greater design flavor and to overlap with the material of courses usually called digital systems design. The courses frequently culminate in student design, usually just a paper design, of some digital system, such as a simple minicomputer. Laboratories and computer simulation languages may or may not be included. Foster (1970), Peatman (1972), and Hill and Peterson (1973) may be used, but the most advanced courses are usually taught from notes or the literature.

Case study computer organization courses are usually preceded by one or more other computer organization courses and concentrate on comparing organizations of several computers. The text, if one is used, is invariably Bell and Newell (1971), but frequently the course is taught from computer manuals and journal articles. The study is usually primarily descriptive although Bell and Newell have contributed to the conceptualization of the subject.

Topical seminars are the least common courses but may become increasingly more important as computer architecture education continues to expand. These advanced seminars usually center on one topic, such as microprogramming or memory organization, for a term or more and are based on discussion of papers from the literature or ongoing research.

### TRENDS

Trends for the future of computer architecture education are hard to predict because the subject depends so heavily on changes in technology. At least for the near future a continued growth of introductory courses seems assured as more engineers, both electrical and other, will need to understand computer organization and machine language programming in order to implement increased numbers of digital systems. Few engineers taking computer organization courses are likely to design computers; hence greater emphasis on interfacing and computer evaluation is needed. Perhaps in the long term the architecture of such systems will have changed to allow implementation by less knowledgeable engineers.

Growth of advanced courses also seems likely as the technology continues to proliferate. As computer architecture matures, the courses will become less descriptive and more conceptual. The decreasing expense and increasing applications of digital components will combine to promote greater emphasis on simplified interfaces with more awareness of users' needs so that, for example, study of computer-assisted instructional systems will emphasize the learner's needs resulting in faster response time, better designed terminals, etc. at the expense of optimal use of components.

### ACKNOWLEDGEMENTS

The author thanks the COSINE Committee for access to the 1972 survey of electrical engineering departments and for other help. The author thanks the referee for his comments. However, the author bears all responsibility for interpretations and opinions.

### APPENDIX

#### ACM 65 - Required Basic Course

##### 2. Computer Organization and Programming

Prerequisite: Course 1 above

Logical basis of computer structure, machine representation of numbers and characters, flow of control, instruction codes, arithmetic and logical operations, indexing and indirect addressing, input-output, subroutines, linkages, macros, interpretive and assembly systems, pushdown stacks, and recent advances in computer organization. Several computer projects to illustrate basic concepts will be incorporated. (ACM, Sept. 1965)

COSINE Recommendations for Computer Architecture Courses

Machine Structure and Machine Language Programming

Content. Computer organization model for interpreting a machine language, machine representation of data and instructions, programming in assembly language, I/O processes, equipment interrupts, stacks, and multiprogramming.

Computer Organization

Content. Elements of a stored program computer, data representation, algorithms for operating on data, arithmetic units, control units, memory units, processor structures, and selected computer examples. (COSINE, Jan. 1970)

ACM 68 - Computer Architecture Courses

Course B2. Computers and Programming

Prerequisite: Course B1

Computer structure, machine language, instruction execution, addressing techniques, and digital representation of data. Computer systems organization, logic design, microprogramming, and interpreters. Symbolic coding and assembly systems, macro definition and generation, and program segmentation and linkage. Systems and utility programs, programming techniques, and recent developments in computing. Several computer projects to illustrate basic machine structure and programming techniques.

Course I3. Computer Organization

Prerequisites: Courses B2 and B3

Basic digital circuits, Boolean algebra and combinational logic, data representation and transfer, and digital arithmetic. Digital storage and accessing, control functions, input-output facilities, system organization, and reliability. Description and simulation techniques. Features needed for multiprogramming, multiprocessing, and real-time systems. Other advanced topics and alternate organizations.

Course A2. Advanced Computer Organization

Prerequisites: Courses I3, I4 (desirable), and I6 (desirable)

Computer system design problems such as arithmetic and nonarithmetic processing, memory utilization, storage management, addressing, control and input-output. Comparison of specific examples of various solutions to computer system design problems. Selected topics on novel computer

organizations such as those of array or cellular computers and variable structure computers. (ACM, March 1968)

REFERENCES

- ACM Curriculum Committee on Computer Science. "Curriculum 68," Communications of the ACM, March 1968, pp. 151-169.
- ACM Curriculum Committee on Computer Science. "An Undergraduation Program in Computer Science - Preliminary Recommendations," Communications of the ACM, Sept. 1965, pp. 543-552.
- Bartee, T. C. Digital Computer Fundamentals. New York: McGraw-Hill, 1966, 1973.
- Beizer, B. The Architecture and Engineering of Digital Computer Complexes. New York: Plenum, 1971.
- Bell, C. G. and Newell, A. Computer Structures. New York: McGraw-Hill, 1971.
- Booth, T. L. Digital Networks and Computer Systems. New York: Wiley, 1971.
- Chu, Y. Computer Organization and Microprogramming. Englewood Cliffs, N. J.: Prentice-Hall, 1972.
- Chu, Y. Digital Computer Design Fundamentals. New York: McGraw-Hill, 1962.
- Chu, Y. Introduction to Computer Organization. Englewood Cliffs, N. J.: Prentice-Hall, 1970.
- Cook, C. C. Digital Computer Instruction in Most Major U. S. Engineering Colleges, Dept. of Industrial Engineering, West Virginia University, Morgantown, April 1963.
- COSINE Committee. An Undergraduate Computer Engineering Option for Electrical Engineering, Washington, Jan. 1970.
- COSINE Committee. An Undergraduate Electrical Engineering Course on Computer Organization, Washington, Oct. 1968.
- Engel, G. "Input from ACM Curriculum Committee on Computer Science," SIGSCE Bulletin, Dec. 1971, pp. 30-31.
- Flores, I. Computer Organization. Englewood Cliffs, N. J.: Prentice-Hall, 1969.
- Flores, I. Computer Software. Englewood Cliffs, N. J.: Prentice-Hall, 1965.
- Flores, I. The Logic of Computer Arithmetic. Englewood Cliffs, N. J.: Prentice-Hall, 1963.
- Foster, C. C. Computer Architecture. New York: Van Nostrand Reinhold, 1970.

- Foster, C. C. Computer Architecture News. Jan. 1973, p. 13.
- Gear, C. W. Computer Organization and Programming. New York: McGraw-Hill, 1969.
- Gschwind, H. W. Design of Digital Computers. New York: Springer-Verlag, 1967.
- Hellerman, H. Digital Computer System Principles. New York: McGraw-Hill, 1967.
- Hill, F. J. and Peterson, G. R. Digital Systems: Hardware Organization and Design. New York: Wiley, 1973.
- Husson, S. S. Microprogramming Principles and Practice. Englewood Cliffs, N. J.: Prentice-Hall, 1970.
- Lewin, D. Theory and Design of Digital Computers. New York: Wiley
- Peatman, J. B. The Design of Digital Systems. New York: McGraw-Hill, 1972.
- Sloan, M. E., Coates, C. L., and McCluskey, E. J. "COSINE Survey of Electrical Engineering Departments," Computer, June 1973z, pp. 30-39.
- Sloan, M. E., Coates, C. L., and McCluskey, E. J. COSINE Survey of Electrical Engineering Departments. Purdue University, 1973b.
- Sobel, H. S. Introduction to Digital Computer Design. Reading, Mass.: Addison-Wesley, 1970.
- Stone, H. S. Introduction to Computer Organization and Data Structures. New York: McGraw-Hill, 1972.
- Ware, W. H. Digital Computer Technology and Design. New York: Wiley, 1963.
- Wiener, N. The Human Use of Human Beings. New York: Avon, undated.



# INCREASING HARDWARE COMPLEXITY— A CHALLENGE TO COMPUTER ARCHITECTURE EDUCATION

R. Hartenstein  
Karlsruhe University  
Karlsruhe, Germany

## ABSTRACT

The paper starts with a survey over history and present-day situation of educational concepts and design methods in computer architecture. Complexity problems, bad design habits, cooperation problems between specialists, as well as their changing range of responsibility are covered, and the consequences of the developmental trends are discussed: now it is time for switching over to an integrated teaching of hardware/software design methods. The HIM scheme (hierarchy of interpretive modules) is suggested as a conceptual machine organization framework for modelling the implementation of language hierarchies. The application of the HIM scheme for better understanding of semantics, and for a derivation of designing guidelines is discussed.

## I. INTRODUCTION

### SOFTWARE COMPLEXITY

About 5 years ago the slogan "software crisis" was coined for a wide variety of problems, caused by increasing complexity and by a lack of design methodology. The most important suggestions to meet those problems rely on imposing restrictions to the freedom of design decisions in a sense to avoid "tricky program structures". The slogan "ego-less" programming has been coined (19) for the virtues of the disciplined programmer, who is needed to meet the software crisis. It becomes apparent, that the software crisis is an educational problem (e.g. see 16).

### HARDWARE DESIGN PROBLEMS

#### The Hardware/Software "Interface Crisis"

The introduction of LSI and decreasing hardware cost more and more give reason for a discussion on the replacement of pieces of software by hardware. By increased utilization of LSI capabilities we are going to face similar complexity problems in the hardware field too. One reason, why the computer architecture community is not yet clearly aware of a "hardware crisis", is the existence of cooperation problems: the lack of mutual understanding between hardware men and software men (one might speak of an "interface crisis" - a subset of the hardware crisis).

#### Changing Ranges of Responsibility

The analysis of changes in the partitioning of design activities among specialists shows, that every new system generation brings us closer to settling the hardware crisis and the interface crisis. Illustration 1 shows the methodological levels of system synthesis (see "hierarchy of levels", chapter I in ref. 1), and the movements of the fields of activities of components/chip manufacturers (C), hardware designers (H), software engineers (S), and language designers (L) in the course

of system generations. The H field of generation 1 covers level 2 thru 7. Unsolved problems in level 2 keep designers from spending much time for levels 3 thru 7. In generation 2 the L field is taking over level 7 and the S field level 6. In generation 3 the S field takes possession of level 5 via microprogramming (survey: e.g. 12, 15), and the C field extends downwards to level 2 by SSI chips. In generation 4 the C field will take over the level 3 via LSI technology, reasonable chip family planning provided (pragmatic definition of "reasonable": ref. 10).

#### Towards an Integration of Design Methods

Design activities in a well formed level 4 mean the use of high level hardware design languages. So design and description methods without the use of any tools from levels 3 thru 1 will be possible, a proper set of register transfer primitives provided (9). This means close similarity of formal tools between S field and H field. So generation 4 is the opportunity for a fusion of S field and H field to form an I field of "integrated design methods" (see Illustration 1). This would settle the interface crisis by integrated design approaches such as "top-down design" (14) or "bottom-up design", as practised in developing some high level language machines (survey: in ref. 5).

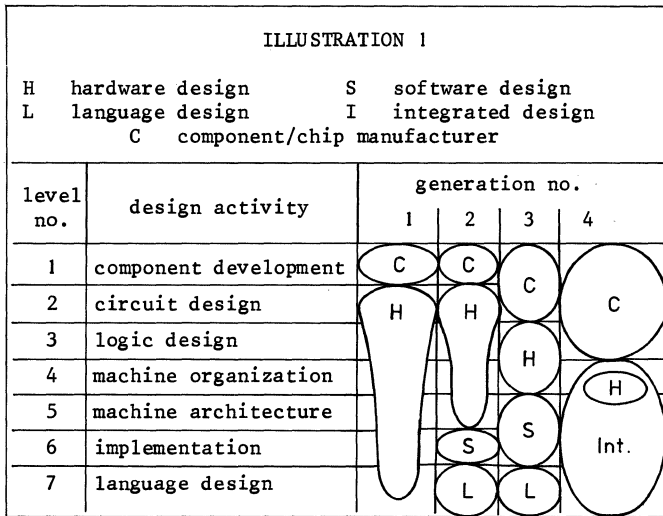
Excellent teaching of integrated design requires a combination of software mens procedural way of thinking and of hardware mens functional and black-box-oriented way of thinking. Descriptive tools for such a methodological combination would be the use of high level programming languages with hardware description features, together with a "grey box" (14) block diagram language (9,10). Illustration 2 shows an example of such a "grey box" block diagram, equivalent to the register transfer statement

```
on t do PA := if AE then PE2 else PE1;
```

where t is a clock signal and AE is a static condition. Such a simultaneous use of a symbolic notation and a grey box diagram in teaching would help to keep aware of the duality of algorithms and their hardware carrier structures. The grey box diagram in this case is no description of a particular hardware component, but it is some sort of low level extension of semantics or something like generalized pragmatics of a language.

#### The "Hardware Crisis"

One symptom of the "hardware crisis" (and of an increasing awareness of it) is an arising discussion on hardware design philosophies, accompanied by a condemnation of a certain kind of traditional hardware structures, which could be called "tricky hardware". ROSIN discovers (but not approves) 4 "rules of thumb", unconsciously used by many machine designers (17), and so causing lots of trouble and inefficiency on the software



side:

Rule 1: "In case of doubt, sacrifice a design concept to preserve cycle time". Rule 1 refers to an unreasonable short-sighted MIPS-squeezing, neither regarding efficiency of the instruction set, nor the hardware/software cost ratio.

Rule 2: "Some facilities are cheap". Rule 2 refers to hardware links not intended in the original plan, introduced for adding "extra features with no extra cost".

Rule 3: "Design constraints don't allow the realization of some otherwise good ideas". The design constraints in rule 3 are those, imposed by an unreasonable set of preconceived components.

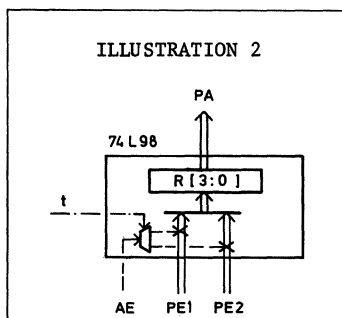
Rule 4: "If it looks nice, it must be beautiful". Rule 4 remembers to features, which are "monuments to the cleverness of the designer" (17).

To meet these problems we need a strong influence on the designing habits in the field of machine organization, aiming at the design of "structured hardware" instead of "tricky hardware". We need the promotion of the virtues of "ego-less" hardware designing. The HIM scheme, presented in this paper, used as a design guideline will be a help in designing "structured hardware".

## THE MODELLING OF DIGITAL PROCESSORS

### Sequential models

The subject of this paper is based on the sequential version of the information structure model on the execution of programs (see chapter 4 in ref. 18), where an information structure model is a triple  $M = (J, J^0, F)$ , with the set  $J$  of information configurations (snapshots), its subset  $J^0 \subset J$  of initial information configurations, and the set  $F$  of operators on  $J$ . The set  $J$  is subdivided by  $J = (C, P, D)$  into a control component  $C$ , a program component  $P$ , and a data component  $D$ , according to illustration 3. Instruction pointer  $ip$  and data pointer  $dp$  are scanning  $P$  and  $D$  under control of  $C$ . The scheme in illustration 3, not being delivered from hardware men, is incomplete for architectural use, as is not showing the embedding of  $F$  into the model.



The embedding of  $F$  is performed by another model, not being delivered by software men, showed by illustration 4, and described elsewhere (e.g. 20,21). The controller"  $K$  combines  $P$  and  $C$  from illustration 3. Block  $F$  contains the resources for the implementation of the set  $F$  of operations.  $F$  and  $D$  are connected by data paths for the transfer of arguments and results. The "order vector"  $Y$  is a selector word for selecting and activating the particular subset of  $F$ , required for the actual step. Status vector  $X$  denotes the feedback from  $F$  to  $C$  for decision purposes.

Automata-oriented Modelling: Some authors model  $K$  separately (e.g. 3,7) and one models the combination of  $K$  and  $C$  (20,21) by finite state machines. It has been demonstrated (8), that these models may be extended into a hierarchy by replacing the model of  $K$  by a finite state transducer (see illustration 5). Thus we get a hierarchical model, according to illustration 6, where a machine  $M_i$  receives instructions in its machine language  $L_i$ . Controller  $K_i$  inside  $M_i$  translates (on-line)  $L_i$  into orders in language  $L_{i-1}$ , the machine language of an inner machine  $M_{i-1}$ . This scheme may be nested to form a hierarchy of machines:  $M_i$  is the inner machine of  $M_{i+1}$ .  $M_{i-1}$  may have an inner machine  $M_{i-2}$  etc.

Programming Language-oriented Modelling: The hierarchy of processes in a program-controlled digital processor implements a hierarchy of languages (e.g. see ref. 13), and does not primarily appear as a hierarchy of automata. One important disadvantage of automata-oriented modelling is, that it fails in modelling the entry of immediate data form instruction streams or statement streams. (By the way: that is an important difference between microprogram control and hardware control). What is needed, is a model, that is more language-oriented. From a hardware point of view, this leads to modelling in terms of interpretations, instead of state transitions.

"Grey Box" Use for Integrated Modelling: "structured programming" techniques for the design language description of computer architectures and machine organizations makes flow charts superfluous. The space made free by throwing out flow charts may be used for "grey box" diagrams. In teaching we achieve by this a better understanding of programming language principles and its semantics, of hardware/software interface problems, as well as integrated design methods. The next section of this paper suggests the HIM scheme as a "grey box" modelling framework for these purposes.

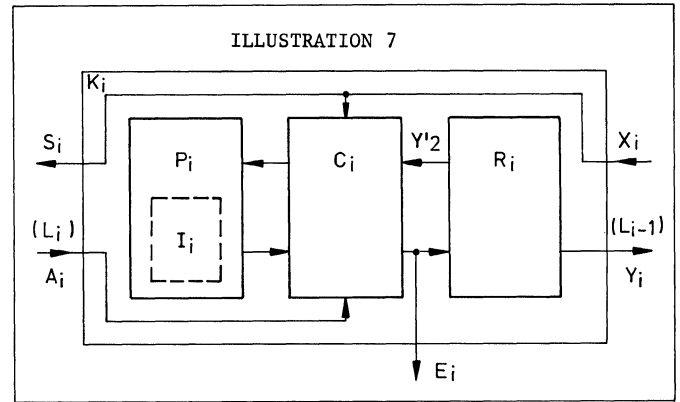
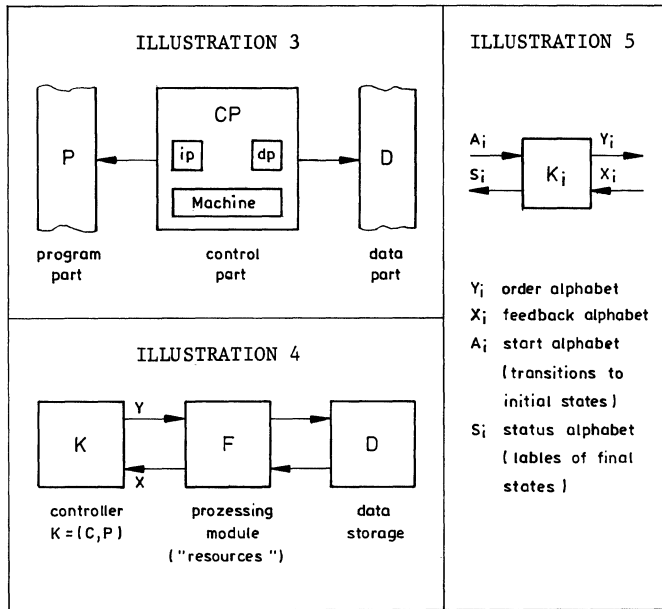
## II. INTERPRETIVE MODULES AS MODELS

### REFINEMENT OF AUTOMATA-ORIENTED MODEL

A model, which may be regarded as a refinement or an implementation of automata-oriented models, can be derived by the fact, that each program execution is subdivided into cycles with the following 3 subcycles (13):

1. The fetch subcycle for the selection of the next element  $l_k \in L$  from the program store  $P$ , where  $L$  is the language  $L = \{l_1, l_2, \dots, l_n\}$ . This selection is performed via adjustment of the instruction pointer  $ip$  (see illustration 3).
2. The recognition subcycle, which performs a test, whether the selected element  $P[ip_j]$  is a legal element with  $P[ip_j] \in L$ , and, which performs the recognition of the specific  $l_k \in L$ , being represented by  $P[ip_j]$ , if legal.
3. The execution subcycle, which performs the proper semantic operations on the data structure  $D$ , as resulting from the recognition subcycle.





$C_i$ ,  $R_i$ ,  $F_i$ , and part of  $D_i$  are the tools for implementing the semantic unit  $F_{i+1}$  of the next higher language level  $i+1$ . As shown by illustration 8, the module  $F_{i+1}$  receives orders via the transfer path  $A_i$  from module  $R_{i+1}$  of the next higher level. After having transmitted an order word via  $A_i$  to  $F_{i+1}$  the module  $C_{i+1}$ , together with  $P_{i+1}$  and  $R_{i+1}$  and all modules of all higher levels, remain in an inactive "waiting state", until via path  $S_i$  an "end-of-execution" message (end) is fed back from  $F_{i+1}$ . Thus the two paths  $A_i$  and  $S_i$  (see also illustration 5) are vertical links to the next higher level of carrier hardware modules. On the other side the two paths  $Y_i$  and  $X_i$  are links to the next lower level modules, formed inside  $F_i$  (also see illustration 5). Thus a hardware-supported block structure (or a grey box structure) of nested interpretive modules, called HIM scheme (hierarchy of interpretive modules) is formed. The above linkage for communication between language levels of different order is called "vertical linkage" or "vertical subroutine linkage" (as this scheme links interpreters like a subroutine calling mechanism).

In a particular level  $i$  within the hierarchy, the subcycles 1 and 2 are performed by controller  $K_i$ , while subcycle 3 is executed by the machine  $M_{i-1}$ . A refinement of  $K_i$  according to subcycles 1 and 2 is shown by illustration 7:  $P_i$  denotes the program store, containing the interpreter  $I_i$ ,  $C_i$  denotes the control module, responsible for the proper sequencing of the stream of instruction words from  $P_i$ , entering  $C_i$  via instruction buffer  $IB_i$ .  $R_i$  denotes the recognition device, having the 2 submodules (not showed here) CL (classifier) and AL (action lexicon). The output of  $R_i$  (produced by AL) is the order vector  $Y_i$ , evoking semantic actions to be performed by machine  $M_{i-1}$ , and the control vector  $Y_{i'}$ , evoking control actions to be performed by control module  $C_i$ .

A second step of refinement is the subdivision of  $M_{i-1}$  into the submodules  $F_i$  (functions) and  $D_i$  (direct data). By both refinements we get a scheme, as showed by illustration 8.  $F_i$  is the implementation of the available set of operators on  $D_i$ . The direct data structure  $D_i$  is a set of registers for temporary storage. The controller  $K_i$ , refined by this scheme, will be called "interpretive module" (IM).

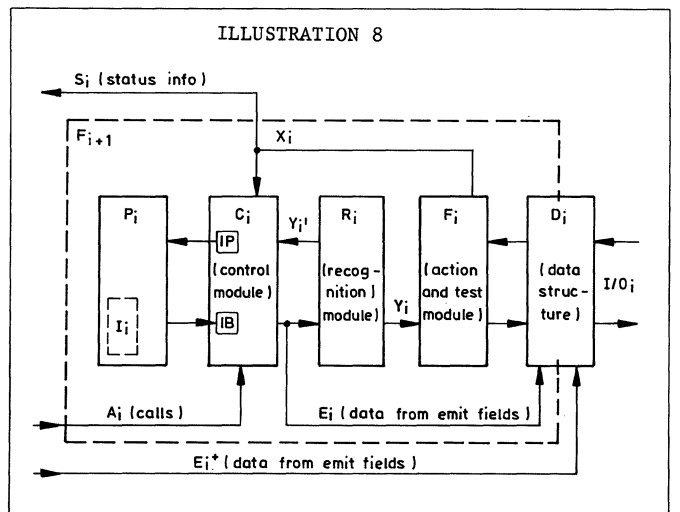
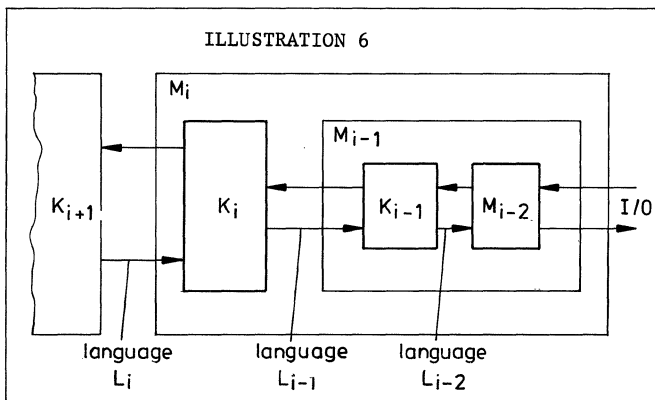
#### LINKING INTERPRETERS TOGETHER TO FORM A HIERARCHY

##### Vertical Linkage

Before the description of signal transfers between the modules in illustration 8 is completed, the synthesis of a hierarchy of such structures is demonstrated (see illustration 9). The  $I_i$  programs in  $P_i$  and the modules

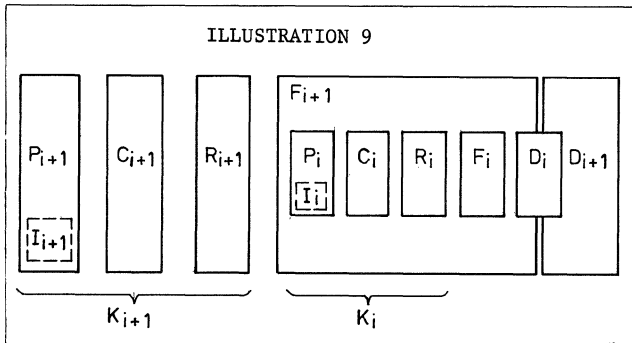
##### Connections Inside One Level

Inside one level of the HIM, there the following transfer paths for interconnecting the modules (see illustration 8). A call via path  $A_i$  causes the transition of  $K_i$  to an initial state via adjustment of  $IP_i$  to the appropriate program entry point of  $I_i$ . Such a call is evoked by the end message of the forerunner program in  $P_i$ , recognized by  $R_i$  as described above. The fetched instruction  $P_i[IP]$ , buffered into  $IB$ , is analyzed by  $R_i$ . The resulting order vector  $Y_i$  is looked up from  $AL_i$  in  $R_i$  and fed to  $F_i$ , evoking the activation of the required subset of data paths in  $D_i$  for the appropriate transfers between registers in  $D_i$  and from emit fields  $E_i$  (in  $IB$ )



or  $E_i$  (from higher levels, if implemented).

I/O to or from  $D_i$  is activated in an indirect manner by placing I/O control messages into special interface registers in  $D_i$ , as for instance a "read"-bit or a "write"-bit, when core storage is external, as in the microprogram level e.g. The control vector  $Y_i$ , derived by  $R_i$  from  $IB_i$  controls via decision logic  $DL_i$  (see illustration 10) inside  $C_i$  the adjustment of  $IP_i$  for the next fetch cycle. The adjustment of  $IP_i$  is executed by the modify paths in MDY in the feedback loop at  $IP_i$ . MDY contains adder, incrementer etc. By path  $X_i$  the following influences on the operation of  $C_i$  are implemented: the cycle time of  $C_i$  via a clock bit in  $X_i$ , the decision for the adjustment of  $IP_i$  by status bits in  $X_i$ , and the request



of the next call from the next higher level by an end-bit in  $Y_{i+1}$ .

Functional Partitioning of resources may be modelled by splitting  $Y_i$  up into subvectors  $Y_i(1), Y_i(2), \dots$  and transmitting them to separate submodules  $F_i(1), F_i(2), \dots$  of module  $F_i$ . Such a partitioning may be modelled in any level. Modelling the submodules of  $F_i$  by the HIM scheme shows several  $K_{i-1}$  modules and module  $K_i$  selectively calling one particular  $K_{i-1}$  module per cycle.

Parallelism and its modelling by the HIM scheme is not the subject of this paper. The modelling of clocked parallelism is trivial and is included into the HIM scheme automatically. The introduction of synchronizing devices for asynchronous parallelism into the model is not impossible and may be achieved by modifying the C module.

Horizontal Linkage or "horizontal subroutine linkage" is the name for a subroutine calling mechanism within the same language level: the call of a subroutine within the interpreter program  $I_i$  (both, the called subroutine, and the calling routine are within the same module  $P_i$ ).

#### THE SUBMODULES WITHIN ONE LEVEL

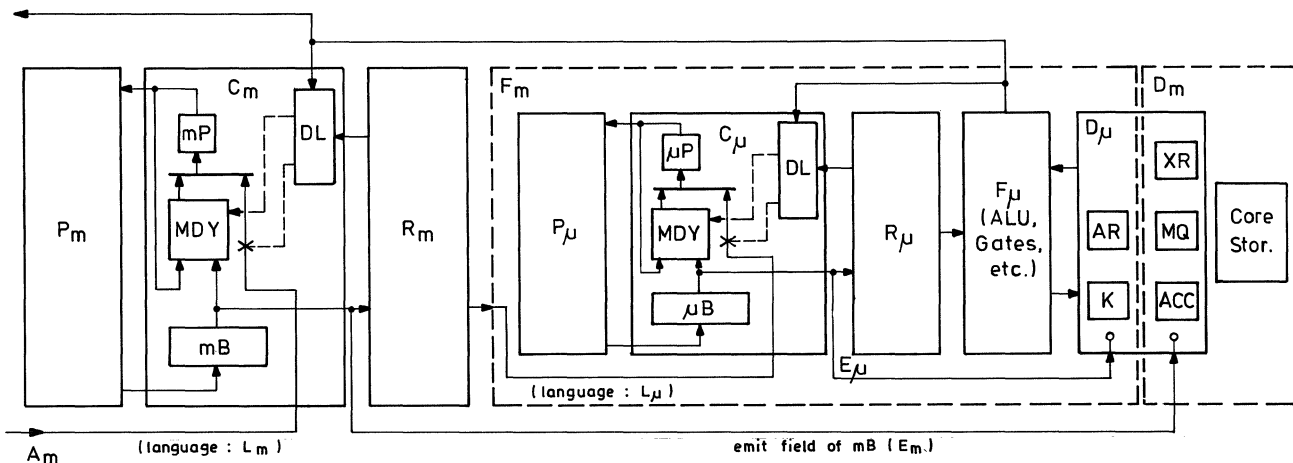
The P module may be a simple program store, when all parts of  $I_i$  are always resident. But the P module may be extended by mechanisms for "load and call" or for "compile, load and call" of non-resident parts of the interpreter  $I_i$ , and may include tables.

The C module may be a relatively simple structure, if no subroutine techniques for  $P_i$  are used. For modelling subroutine techniques  $IP_i$  is extended into a pointer stack  $IPS_i$ , and in the case of an internal subroutine call the request of an order via path  $A_i$  is replaced by push operation on  $IPS_i$ .

In some cases "immediate orders" are entering  $C_i$  via  $A_i$  (e.g. the  $C_\mu$  module of the DEC pdp-8 machine, during the processing of microcode from special machine instructions). In this case  $IB_i$  receives its input directly from the  $A_i$  input, and not from  $P_i$ . The decision logic DL has to provide a tag sensing feature for recognizing "immediate" orders entering  $A_i$ . If we extend  $IB_i$  into a stack  $IBS_i$  for pushing immediate orders and use another stack in the  $D_i$  module, we are able to model the direct evaluation of expressions, such as illustrated by the shunt station model.

The R-module may be a decoder network, when used for modelling the decoding of a machine instruction or a microinstruction.  $R_i$  may be implemented in a more sophisticated manner and sequentially, when constructs of a higher level language with a wide variety of sentence formats have to be analyzed. The decoding of instructions of byte-oriented machines is an example of decoding variable length objects in parallel. In ref. 9 is demonstrated, how the equivalence of sequential networks and combinatorial networks can be used for a uniform modelling of decoder networks and a class of parsing algorithms in terms of a set of register transfer primitives. This idea allows a uniform modelling of a wide variety of R-modules from different levels of a complex digital systems. So the AL-submodule of  $R_i$  may be a table (in higher levels),

ILLUSTRATION 10



a wiring scheme (in lower levels) or even a model for the pulse phase level below the register transfer level, which is useful for pedagogic purposes and for the analysis of the behavior of certain register transfer structures without using formal tools of the level of logic design (10).

The F-module is a combination of all transfer carriers, used for semantic purposes, such as gated transfer paths, paths from and to registers for the implementation of register assignment operations, and transformational Transfer paths, such as arithmetic and logic units, when it is used for modelling  $F_i$  in the micro-program level. In higher levels the F-module appears as a combination of abstract transfer carriers, yielded by omission of intermediate transfer steps, implemented in lower levels.

The D-module is the set of all data containers, such as read/write registers and read-only registers (constants or input terminals from emit fields), which are directly or implicitly addressable by constructs of the language, used for  $I_i$ . Those data containers, which are addressable by language of  $I_i$  only indirectly, are parts of the external data structures, called I/O in illustration 9. Those data containers are directly or implicitly addressable only within one of the higher levels of the hierarchy. Those registers, which are not at all addressable by the language of  $I_i$ , may be addressable within lower levels of the hierarchy, not modelled in the present level. Sometimes there are registers, which belong to 2 (or more?) levels' D-modules simultaneously, as for instance the accumulator register or sometimes other general registers, addressable by  $L_m$  and by  $L_\mu$  and thus belonging to  $D_m$  and  $D_\mu$  simultaneously (see illustration 10). In such a case  $D_m$  and  $D_\mu$  are overlapping, as e.g. shown in illustration 9 and 10 (here m stands for "machine language" and  $\mu$  for "micro language").

### III. POSSIBLE APPLICATIONS OF THE HIM SCHEME

#### MODELLING EXISTING SYSTEMS

Illustration 10 demonstrates the pedagogic use of the model for structuring the register transfer carriers of a microprogrammed instruction set processor by modelling it into a 2 level's scheme. The HIM scheme also is useful for modelling more than two levels, and, for modelling higher language levels. The embedding of compiler and assembler software into the framework of the HIM scheme will be possible by using 2 different schemes: one scheme for compile time modelling, and one scheme for run time modelling.

#### Classification of Architectures in Terms of the HIM

All digital systems are implementations of language hierarchies. A very useful criterion for classifying computer architectures is the degree of directness of the hierarchy implementation. It is more direct in pure interpretive systems, than it is in compiler-oriented systems. It is the more direct, the more language levels have C modules and R modules, being implemented in hardware, instead of software. Virtual (software-implemented) C and R modules require a time-sharing of lower levels' C and R modules, and in many cases of F and D modules too. In conventional instruction set processors, for instance, the higher language levels share with the machine language level in the interpretive module of the latter level. Let me classify this as "tricky" implementation of hierarchy and call it "vertical multiple use". It seems practical, however, to classify "horizontal multiple use" (subroutine mechanisms within the same language level) not as "tricky".

Let me give further definitions of hardware structures. "Structured hardware" is the basis for a direct implementation of a hierarchy, and an indirect implementation of a hierarchy of languages results in hardware, which is less "structured". Another criterion of the directness of an implementation is the existence of vertical multiple use of working stores and its accessing hardware for P and D modules. At least in this respect the "stored-logic machine" and the "von-NEUMANN machine" show a considerable degree of indirectness in hierarchy implementation.

#### Trends towards Structured Hardware

In the 70ies there is a tendency for transferring more and more complexity from the software part to the hardware part of systems. This is demonstrated by the successive advent of the following classes of architecture (5):

1. von NEUMANN-type architecture
2. syntax-oriented architecture (e.g. the B 5500)
3. IHLL-architecture (indirect high level language, survey: ref. 5, also see ref. 11)
4. DHLL-architecture (direct HLL (5, also: 2,6))

This sequence of architectures demonstrates the following developmental trends in system concepts research:

1. Increasing directness in implementing language hierarchies, which means increasing similarity to the HIM scheme.
2. The increasing complexity of hardware makes it more and more advisable, not to produce tricky hardware, but to produce structured hardware instead.
3. Because of low hardware cost, the extremely efficient utilization of particular hardware submodules is no more a relevant design objective. We now can afford idling modules. This leads to more hardware instead of tricky hardware.
4. A growing tendency to functional partitioning of hardware (e.g.: ref.5) yields a tendency to more structured hardware.

These developments make systems more and more appropriate for being modelled by the HIM scheme, and the use of integrated design methods, as e.g. supported by "grey box" modelling via HIM scheme.

#### Straight-on Teaching of Computer Architecture

For the use as examples for teaching computer architecture we have now available: considerable know-how out of papers on DHLLP design (bibliography: 5), as well as knowledge and teaching experience on conventional architectures. The question arises: which type of examples gives us more benefit in understanding programming language principles and semantics, in the linkage between language levels, in computer architecture and machine organization? The author believes, that for the student the direct implementations of hierarchies are less opaque, than indirect hierarchy implementations. The introduction into the "tricks" of indirect implementations would be better scheduled as a second step, after a successful teaching of direct implementations, relying on "structured hardware"-type hypothetical (or real) architecture examples.

### IV. CONCLUSIONS

For the philosophy of straight-on teaching, the HIM scheme has been proposed as the framework for a hypothetical direct architecture example, as a "grey box"

model for teaching integrated hardware/software design, and (with some restrictions for economical reasons), as a guideline for teaching structured hardware design, in order to avoid cumbersome tricky hardware and so to promote the virtues of the "ego-less" system design.

#### ACKNOWLEDGEMENTS

The author would like to express his appreciation to Profs. W. Görke and D. Schmid for encouragement to work on this subject and for critical and constructive comments, and to Profs. R.F. Rosin and S. Wendt for valuable discussions.

#### BIBLIOGRAPHY

1. Bell, C. G., Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, New York 1970
2. Bjørner, D., "On the Definition of Higher-Level Language Machines", Proc. Symp. on Computers and Automata, New York 1971, Polytechnic Press, Brooklyn, N.Y. 1971
3. Cascaglia, G.F., Gerace, G.B., Vanneschi, M., Equivalent Models and Comparison of Microprogrammed Systems, Internal Rep. 3, Spec.Ser.Conv. CNR-ENI, CNR Pisa 1971
4. Chu, Y., Computer Organization and Microprogramming, Prentice Hall, Englewood Cliffs, N.J. 1972
5. Chu, Y., Introducing to High-level-language Computer Architecture, TR-227, Univ. of Maryland. Comp. Sc., Feb. 73
6. Frick, A., Ein Rechner mit Problem-orientierter Maschinensprache BASIC, Informatik-Kolloquium, Karlsruhe 1973
7. Glushkov, V., "Automata Theory and Formal Microprogram Transformations", KIBERNETIKA, vol. 1, no. 5 (1965)
8. Hartenstein, R., Über die Schnittstelle zwischen Hardware und Software, Informatik-Colloquium, Hamburg 1971
9. Hartenstein, R., "A half-baked Idea on a Set of Register Transfer Primitives", SIGMICRO News1. 4/2, 1973
10. Hartenstein, R., "Towards a Language for the Description of IC Chips", SIGMICRO News1. vol. 4, no. 3 (1973)
11. Hassit, A., Lageschulte, J.W., Lyon, L.E., "Implementation of a High Level Language Machine", Comm.ACM vol. 16 (1973), no. 4
12. Husson, S.S., Microprogramming - Principles and Practices, Prentice Hall, Englewood Cliffs N.J. 1970
13. Lawson jr., H.W., The Changing Role of Microprogramming, ACM Seminar Course Readings, 1972
14. Lipovski, G.J., "A course in Top-down Modular Design of Digital Processors", Workshop on Education and Computer Architecture, Atlanta, Ga., august 1973
15. Rosin, R.F., "Contemporary Concepts of Microprogramming and Emulation", Comp. Surveys vol. 1, no. 4 (1969)
16. Rosin, R.F., "Teaching about Programming", Comm. ACM vol. 16, no. 7 (1973)
17. Rosin, R.F., "The Significance of Microprogramming", ACM Intl. Computing Symp., Davos, sept. 4 - 7, 1973
18. Wegner, P., "Data structure Models for Programming Languages", SIGPLAN Notices vol. 6, no. 2 (febr. 1971)
19. Weinberg, G., The Psychology of Computer Programming, van Nostrand Reinhold, New York N.Y. 1971
20. Wendt, S., "Eine Methode zum Entwurf komplexer Schaltwerke unter Verwendung spezieller Ablaufdiagramme", Elektron. Rechenanl. vol. 12, no. 6 (1970)
21. Wendt, S., "Zur Systematik von Mikroprogramm-Strukturen", Elektron. Rechenanl. vol. 13, no. 1 (1971)
22. Wendt, S., Entwurf Komplexer Schaltwerke, Springer-Verlag Berlin/Heidelberg/New York 1973

# REVIEW OF THE WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION

George Rossmann  
*Palyn, Inc.*

## Abstract

This paper reviews the presentations and discussions of the participants in the Workshop on Education and Computer Architecture held in Atlanta, Georgia on 30 August 1973.

## I INTRODUCTION

A workshop on education and computer architecture was held on 30 August 1973 in Atlanta, Georgia. It was cosponsored by the ACM Special Interest Group on Computer Architecture (SIGARCH) and the IEEE Computer Society Technical Committee on Computer Architecture (TCCA).

The goal of the workshop, as stated in the invitation to participants, was to develop the foundations for a series of courses that would provide a good education in computer systems design. What we had in mind was the detailed specification of two or more courses in computer architecture which would replace both the computer organization course developed by a COSINE TASK FORCE (3) and the organization courses (13 and A2) outlined by the ACM Curriculum Committee on Computer Science (1). However, the contributions received from the workshop participants addressed such a broad perspective of computer systems design that we ended up discussing most of the elements of a typical computer engineering curriculum (4). The workshops' observations about current programs in computer engineering education and its proposals for introducing some fresh ideas into these programs are the primary reason for circulating this summary of our discussions.

To insure a broad perspective, participants were invited from universities and computer manufacturers. They were asked to contribute in whatever way they could. We solicited papers for oral presentation as well as position papers from those invitees who were unable to attend but still wanted to contribute their ideas. The result was a workshop attendance of twenty-four. There were ten formal presentations given and three position papers which were summarized.

The program was as follows:

### WORKSHOP ON EDUCATION AND COMPUTER ARCHITECTURE

#### Program

#### 9:00 a.m. - 12 Noon

- W.J. Watson, "What is This Thing Called Architecture, and Where is it Going?"
- R.A. Dammkoehler, "Experimental Modular Machines"
- D. Siewiorek & J. Grason, "Using Register Transfer Modules (RTM's) in Teaching Computer Architecture"

- R. Ashenhurst, "Hierarchical Systems for Laboratory Automation"
- R. Rosin & B. Shriver, "Towards Reasonability in CPU Design: A Case Study"
- T. Rauscher, "The Influence of Specific Problems on Machine Architecture"

#### 1:00 p.m. - 5:00 p.m.

- F. Brooks, "Computer Architecture Education"
- S. Fuller, "An Annotated Reading List for a Topics-Oriented Course On Computer Structures"
- C. Hooper, "Study of Computer Architecture Through Simulation"
- G. Lipovski, "A Course in Top-Down Modular Design of Digital Processors"
- H. Hellerman, "New Emphasis in Computer System Education"
- H. Lorin, "Operating Systems Education"
- F. Hill, Position Paper on "Digital Systems: Hardware Organization and Design"

The workshop presentation and discussion can be approximately partitioned into three main topics: Discussion of the educational methods used to build computer systems design experience, specification of the structure and content of some courses to be included in a computer architecture sequence, and finally, description of some of the elements of computer science education which are considered to be essential in the training of a computer architect.

## II Educational Methods

Several strategies for teaching computer systems design have emerged: simulation, modular systems design, the case study approach, and theoretical studies. The appropriateness of each of these depends upon the objectives and the level of the instruction.

Simulation is well accepted as a means for studying computer structures (3). Its use was discussed by C. Hooper. In conjunction with a senior-level course on computer architecture, he offers an extensive software laboratory component in which students are required to design and program a simulator and cross assembler for some well defined computer system in order to study its characteristics and evaluate its features. This requires an enormous amount of time. The student increases his programming experience and finishes the

course having examined at least one machine architecture very closely. This is probably sufficient accomplishment for an undergraduate course, but it would not be appropriate for developing the breadth of understanding needed by a professional computer architect. It was suggested that his needs might be better served in other ways. For example, he might be required to program a few problems on a set of machines to get a feeling for the effect of architectural decisions on problem processing. Or he might work with a single machine and a single problem and determine the effect on the program for that problem caused by deleting certain features of the machine.

Modular systems introduce an efficient experimental dimension to computer systems design. The October 1973 issue of *COMPUTER* surveys the state of their development, describes experiences various groups have had in using them, and offers some conjectures on why designers and users are attracted to them. At the workshop, Fred Dammkoehler and Dan Siewiorek discussed the impact of modular systems in studying computer architecture.

Dammkoehler argued that in engineering effective modeling of real phenomena is the key to understanding them. He suggested that, in the context of computer engineering, modular systems which lend themselves naturally to model construction and manipulation as well as allowing students to maintain contact with reality serve the learning process best. The ways in which macromodules accomplish this were observed in the context of a comparative architecture study conducted by his graduate students. Experiments were undertaken in order to determine the extent to which the efficiency of a structural analysis procedure could be improved by systematically increasing processor concurrency. Three processor structures, serial, asynchronous and pipelined, and a hardware monitor were built within a reasonably short period of time and the analysis procedure was executed on each of the processors. Good results were obtained indicating that by using a set of appropriately designed modules it is possible in "reasonable" amounts of time to explore a number of design alternatives.

Siewiorek described similar experiences with the use of Register Transfer Modules (RTM's) in studying computer architecture. A series of laboratory exercises of increasing complexity; design of a desk calculator, display processor, simple computer, pipeline processor, etc., were used to develop computer systems design experience. What is remarkable about this laboratory is that it is used in conjunction with a junior-senior level Computer Systems course.

Modular systems make it possible to consider relatively complex digital systems at a functional level where they can be handled by the electronically naive, and they make the design and implementation of such systems quick and understandable. The modular approach results in extraordinary student motivation. Laboratory experiences in which students can work in close co-operation to build systems which solve real problems are becoming much more significant. Simulation approaches fail to develop that intense commitment.

Jack Lipovski presented the description of a course for designing digital systems in an MSI and LSI hardware environment. Commercial chips are used as modules. They are incorporated into a design by deducing from the statement of the problem via a high level hardware design language what modules are required and how they should be interconnected.

The case study approach addresses the upper levels of computer systems design: computer architecture, which defines the attributes of a computer system as seen by the programmer; and physical implementation, which includes the organizations of processors, memories, switches, input-output devices, etc. There was no dispute with the proposition that at the graduate level the proper study of the machine designer is machines and that there is no substitute for close examination of other designers' machines. Fred Brooks, Sam Fuller, and Joe Watson discussed this strategy.

Professor Brooks point of view was especially interesting. He distinguishes computer architects from computer engineers. In his view, computer engineers are responsible for implementation and computer architects are responsible for the principles of operation manuals. Further, he characterizes training in computer architecture as elementary and advanced. The elementary level is designed to teach every computer scientist how machines work and to help him understand the forces that led to the design decisions which he has seen reflected. Any competent computer science instructor can teach it from the literature. The advanced level is designed to teach the 5 to 10 professional computer architects who are needed by industry each year how to come up with a master plan. This training, he asserts, can only be offered by experienced and practicing computer architects. It cannot be acquired from the literature and there is not substitute for learning it from someone who has been through the design experience of a real machine.

### III A Basic Course

There are three aspects to Brooks' course on computer architecture. The first is an in-depth analysis of a two dimensional computer space like that shown in Figure 1.

The first four topics form the fundamentals of computer architecture; although, Tom Rauscher pointed out that the design of special purpose machines to solve specific problems may make other aspects of computer systems designs more significant. The computer systems are ordered to preserve their evolutionary development. The space is scanned in both directions in parallel. The horizontal dimension compares different techniques for solving classical problems and charts their evolution. The vertical dimension shows how solving one of the problems in a particular way constrains the solution of another problem to be less than best. Studying individual machines in the absence of such a typical framework; i.e., a pure case-study approach, frequently produces unsatisfactory results.

The challenge in analyzing systems is to figure out what the designers' reasons for their decisions were. The space helps to discover the rationale behind some of these decisions. The reasons for the remaining ones may not even be technical and impossible to deduce.

Bob Rosin and Bruce Shriver discussed some of these subtleties which accompany design. It is their contention that some of the decisions made by computer systems designers and implementers are unreasonable. They are based on invalid rules of thumb, a narrow and incomplete view of the ultimate users' needs, and the use of inappropriate tools. Their paper presents a case study of a real machine design and shows how it was possible to make a somewhat reasonable system out of a potentially unreasonable one without sacrificing anything other than some traditionally held bad ideas.

Figure 1

		Computer Systems								
		IBM 701- 7094	UNIVAC 1101- 1110	IBM 650- 7074	IBM 1401- 7010	STRETCH	B5000 KDF 9	MU5	PDP11	etc.
Computer System Design Problems	Representation									
	Addressing									
	Operations									
	Sequencing									
	I/O Control									
	Arithmetic									
	Character & Bit Operations									
	Memory Hierarchies									
	Microprogramming									
	etc.									

In light of this, the student's problem is not only to seek the reasons for decisions, but he must try to assess their ultimate consequences. The teacher should provide as much real frequency data; e.g., opcode distributions, branching conditions, etc., as he can locate to support the student's analysis. Ultimately, the student has to develop an intuitive sense for the dynamics of machine activity.

The second aspect of Brooks' course is a complex software laboratory project based on a real industrial or university need. The goal of such a project is not to learn how to program, but, just as in the case of modular systems, to learn to work as a member of a team which must build, debug, document, and demonstrate something on a schedule.

The final aspect involves reading classical papers. Sam Fuller offered an annotated reading list in which student reaction to many of these papers is surveyed. Statistics were gathered on responses to five questions: Did you read it?, clarity, detail, understandability, and value. Fortunately, many papers are available in the text by Bell and Newell (2).

Some comments suggested that an equally valid approach to teaching a course like this could be developed from an implementation driven point of view.

#### IV Essential Supporting Courses

An essential part of any computer system is its operating system. The architecture and implementation of a machine cannot be separated from the operating system which runs on it, since together they constitute the environment which the user sees. A course in *Operating Systems Principles is an essential companion to the course in computer architecture (5)*. Otherwise, as Sam Fuller pointed out, the rationale behind various architectural decisions and the reasons for including certain topics; e.g., virtual memory, in computer architecture courses would be incomprehensible to those students who have no operating system experience.

Hal Lorin addressed the fundamental problem of operating systems education directly. The problem is

that there is so much material and so many paths through the material that it is almost impossible to manage or defend a single unified approach to design or discussion. The working solution to the problem has been to fragment the material into separate disciplines, to design elegant internal structures without comprehending or investigating their impact on the user, to concentrate on a few problems of current interest to the exclusion of others, to represent the structure of a single system as a natural structure for all systems, to exclude historical material, and to avoid the full implications of systems use. All these prevent the computer architect from achieving significant insights into the dynamics of the relationship between operating systems, computer architecture, and the computational environment. The fundamental need in education therefore is to create an appreciation within the computer architect for the essential unity with which a user sees his system, the various ways in which his machine will be used, and how the user judges the success or failure of his system.

This last point was expanded by Herb Hellerman. He argued that a consciousness of system performance, which he defined as an assessment of the qualities and features of a system by as objective a set of measures as possible so as to judge its ability to process work for some end and to compare it with other systems, should permeate throughout the computer science curriculum. By using objective and quantitative measures of performance as much as possible and qualitative measures when real measures do not exist, a student ought to be able to develop some sense of the worth of a computer system. If he combines these with cost data, he should be able to evaluate the cost effectiveness of a system. Joe Watson also represented this process as fundamental from the manufacturer's point of view.

#### V Conclusions

Based on what the workshop considered, it might have been more appropriately entitled Workshop on Computer Engineering Education. It clearly demonstrated the growth and interdependence of the subjects recommended for a computer engineering curriculum in (4).

Of particular interest is the emphasis now being placed on the laboratory experience, both hardware and software, which should accompany these upper-level courses.

A special task force is now being constituted to realize the original goal of the workshop; the development of new course descriptions for computer architecture courses. The suggestions and contributions made by the participants in this workshop should make that effort simple and fruitful.

#### Acknowledgements

The author wishes to acknowledge the assistance of Allan Marcovitz and Willis King. They helped provide much of the energy and work required to make the workshop a reality.

#### BIBLIOGRAPHY

1. ACM Curriculum Committee on Computer Science. Curriculum 68, Comm. ACM 11,3 (Mar. 1968), 151-169
2. Bell, C.G. and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill Book Co., 1971
3. COSINE Task Force II. An Undergraduate Electrical Engineering Course on Computer Organization, Oct. 1968
4. COSINE Task Force Report IV. An Undergraduate Computer Engineering Option for Electrical Engineering, Jan. 1970
5. COSINE Task Force VIII. An Undergraduate Course on Operating Systems Principles, June 1971



# MICROMODULES: MICROPROGRAMMABLE BUILDING BLOCKS FOR HARDWARE DEVELOPMENT

Richard G. Cooper  
National Security Agency  
Fort Meade, Maryland

## I. Introduction

The algorithm design phase in the development of special purpose hardware is usually a very small part of the overall effort. A much larger portion - often 90% or more - is expended on logic design, fabrication, and debug. Furthermore, since the pure algorithmic complexity of hardware tends to be small, algorithm design errors typically account for a small part of the total debug time; errors due to electrical effects consume the lion's share. Precautions taken during machine design, fabrication, and debug to minimize reflection, switching noise, and synchronization errors are time consuming and expensive. This situation is at its worst when various types of equipment are to be produced in small quantities. As the use of Schottky-TTL and ECL increases, the problem will become more severe.

The purpose of the Micromodules project is to greatly reduce the amount of effort expended on logic design, fabrication and debug for small quantity developments. Secondly, with this modular approach, a quick reaction capability is sought that would allow a large reduction in the time interval between system specification and the delivery of the finished product. Finally, by simultaneously simplifying and speeding up the development process, we aim to improve the practicability of implementing more complex equipments.

These goals can be achieved by the development of a family of microprogrammable modules. Each module will be architecturally compatible with a small class of common hardware structures with obeisance to a standardized interconnection discipline. The system designer will obtain a collection of modules from inventory and configure them, by means of the interconnection discipline, into a system which is architecturally suited to solve the problem at hand.

It is likely that many systems will require some special hardware development in addition to the standard modules; our intention is to minimize the quantity and complexity of such special equipment. As the project progresses, additional common structures will be identified and the family of micromodules will be expanded to contain them when justified.

Our approach is not without precedent; the Macromodules project [1,2,3] at Washington University has been a fundamental source of inspiration. There, under the direction of W. Clark and C. Molnar, a set of asynchronous building blocks were constructed. These can be interconnected with standard cables. Loading factor allowances, noise attenuation and techniques for synchro-

nization were built into each module. Functionally, their modules are quite simple. Using adders, registers, memories and other modules of similar complexity, they can construct systems of interconnected blocks which are effectively free from electrical errors. System implementation can be accomplished quickly and easily; it is not uncommon for an engineer to design, construct, and debug a significant system in a matter of days.

Due to the functional simplicity of each module, the relative cost of eliminating intramodular electrical errors is high. However, macromodules are intended for the construction of experimental equipment. A number of modules are configured to implement a certain algorithm; the system is used for a short period of time and the modules are then returned to the stockpile for later use. In such an environment, the cost of each module is not very important. It will be used in many different implementations and only a fraction of each module's cost need be attributed to each use. The time and effort required to build each experimental system is the more important consideration.

Our approach has been to apply the macromodular concept to the development of unique operational special purpose equipment. In this environment, the cost of each module is quite important; it will be used in only one machine; therefore, the relative cost per module of eliminating electrical errors must be reduced. To achieve this reduction, we chose to increase the functional power of each module rather than relax the interconnection discipline. A given algorithm would be implemented with fewer, more powerful modules; as a result, the overhead of eliminating electrical errors is reduced.

## II. Microprogrammed Machines

Note that a more complex module increases the danger of sacrificing the flexibility required for constructing special purpose hardware of greatly varied designs. If flexibility is to be retained, individual types of modules should be modifiable within the range of their architectures to suit a diversity of applications. For this reason, our modules are often microprogrammed, i.e. designed with alterable control memories. Integrated circuit PROMs (programmable read-only memories) will be used to specify the functions to be performed by each module. When new applications of existing module architectures are required, new PROMs will be designed to tailor the modules to the application. With this approach, we can, in effect, create a wide variety of complex building blocks for a minimum of

developmental effort.

Most projects will still require some ROM design. Although many data routing and formatting functions will be satisfiable with basic designs, it is not likely that needed processing and sequencing functions will have been previously designed. Therefore, a system built with the macromodular approach, a system built with micromodules will require more effort - weeks instead of days. Nevertheless, the effort involved in system implementation will be greatly reduced when compared with that of current, traditional hardware development methods.

The total cost of a given implementation will probably also be reduced. Cost reductions will be achieved in three areas. Since effort can be translated into dollars, substantial savings will be gained in reducing total effort. Because the modules will be produced in quantity, the economies of scale create further savings. Finally, the extensive use of MSI and LSI technology, usually unjustifiable in one-of-a-kind equipment development, also contributes to overall economy. Offsetting these reductions, several factors require expenditures not normally accruing to equipment development. The cost of developing the modules, their associated production tooling and inventory maintenance costs must be distributed among the equipments produced. Any portion of each module's capabilities that is not effectively used in a given equipment must still be purchased. Quantitative comparisons of these factors cannot be made at this time, but it appears that the overall cost per equipment will be reduced.

In order to clearly describe the micromodular approach, we must examine those user-microprogrammable machines currently on the commercial market.

The great majority of commercial machines are oriented towards emulation; for this reason, they tend to be complex and expensive. Because the machines will be used in stand-alone configurations, the architectural emphasis tends toward high speed full word arithmetic and logical processing overlapped with random access memory fetch. Very limited Boolean capabilities and almost no multiple Boolean decision and control functions are included. When used in special-purpose equipment, emulation machines require considerable interface logic. Relatively small amounts of local high speed storage are common, because main memory offers large amounts of cheaper, slower storage. Due to the complexity of emulation machines, their cost prohibits multiprocessing systems for many hardware applications. Even when multiprocessing is used, the burden of synchronization falls on the microprogrammer, or hardware synchronization must be provided.

Another large segment of the commercial market is directed towards the implementation of disk and tape controllers. Few of these are truly user microprogrammable and virtually all are fixed architecture machines. Synchronization of multiple machine configurations must be microprogrammed or implemented by means of additional hardware.

Recognizing the limitations of current machines, we decided to use a building block approach with the micromodules. Each module is designed to solve a small class of common hardware problems, without frills. The

emphasis is on low cost, high instruction cycle rate, and the possibility of cooperation between modules. Although the class of problems compatible with each module's architecture is small, several modules can be configured to achieve the requirements of a given implementation.

### III. Modular Design Considerations

The separation of functions is an important theme in the design considerations. Since microprogramming can be a difficult task, the separation of functions is useful in dividing the problem into subproblems which can more easily be solved. Each subproblem can then be attacked using the most appropriate module. As new classes of subproblems are identified, new modules tuned to these classes can be developed. System debug can also be simplified by the subproblem approach; each subsystem can be debugged individually, postponing debug at the system level until the last subsystems are ready.

Since systems will be constructed from collections of modules, synchronization and buffering are also important considerations. Facilities for synchronization and buffering are built into each module in hardware. In most practical cases, loop-free networks of modules can be constructed, freeing the designer from these problems. Where loops must be constructed, some simple precautions will ensure that no deadlock problems exist. As will be shown later in this paper, a minimal amount of programmed synchronization can greatly improve efficiency for certain kinds of processes.

Connections between modules can be either arithmetic or Boolean. Arithmetic paths are eight bits wide (one byte). Each byte path is constructed by connecting a polarized ten-conductor cable between a byte output port on one module and a byte input port on another. Each port maintains a FULL flip-flop which specifies whether the port contains data. When data is transferred from an output port to an input port, the FULL flip-flop in the output port is cleared and the FULL flip-flop in the input port is set. The transfer of data between ports and control of the FULL flip-flops during transmission are performed completely in hardware. Two wires in the ten-conductor cable are used for handshaking signals. Transfer between ports is accomplished by logic built into each port. Since each port contains its own data buffer register, the interconnected modules can be performing computations while the transfer is taking place.

Synchronization of byte data transfers with processing is accomplished by use of the FULL flip-flops. If a microinstruction attempts to read data from an input port which does not contain data, completion of that instruction is suspended until data is transferred into the port by the handshaking logic. When an input port is read, its FULL flip-flop clears, allowing the handshaking control to transfer in another byte. Thus each access of an input port reads a new byte of data regardless of the input arrival rate. Similarly, a microinstruction that attempts to place data into an output port, which already contains data, is suspended until the

port empties. Since both input and output ports contain data storage registers, all byte transfers between modules are double buffered by the hardware.

Each arithmetically oriented module can contain multiple input and output ports for byte data. Thus data words larger than eight bits can be transferred serially by byte or in parallel along several cables. Parallel transfers occur independently. Since modules can be processing while transfers take place, and since data is double buffered, the duration of data transfer can be several instruction times long without much degradation of performance. This relatively slow data transfer, combined with fixed loading factors and reflection characteristics, reduces electrical interconnection errors to a low level. Resistor terminators are built into the input ports and interconnection cables are shielded.

Boolean interconnections are of two kinds: level signals and pulsed signals. Level signals are useful for connections between the modules and peripheral equipment. Level signals can be used for controlling and sensing Boolean lines, e.g. tape and disk drives. Pulsed signals are useful for synchronization tasks within the network of modules.

Coaxial cables are used for transmitting Boolean signals and each module can contain one or more Boolean input and output ports. Switches are provided on some modules to specify whether a port will be a level or pulsed signal device.

Level signals are strobed into flip-flops at the beginning of each instruction cycle to assure unambiguous operation. Schmidt triggers are used in some modules to perform level conversion and signal conditioning.

Pulsed signals require a rise and fall cycle of operation. A two phase flip-flop configuration is used on the input lines to synchronize pulsed signal transmission. A received pulse is stored in a flip-flop until the receiving module tests that flip-flop. When a pulsed flip-flop is tested, it is automatically reset. Pulsed signals are therefore not acknowledged in hardware by the receiving device. If a given system requires acknowledgement, this task must be performed in firmware.

#### IV. Design Aids

The design of ROMs, as has been previously stated, is a difficult task. For many projects, ROM design will be the most time consuming part of system implementation. For this reason, numerous ROM design aids are planned. Design aids will be written in time-sharing Fortran IV for the DEC PDP-10.

A basic table-driven assembler will be constructed. Individual symbolic assemblers can then be written for each module by providing the basic assembler with the proper tables.

A single preprocessor program will be used to expand macro routines prior to assembly. Alphanumeric text, consisting solely of macro control statements, will be input to the preprocessor. Expansion will then be independent of the individual assembly languages; thus the macro capability need not be provided for every version of the

assembler. ROM designers must expand macro calls individually and then edit the expanded macro text into the body of the program.

A functional simulation of each module will be provided. The ROM designer can then debug his microprogram by repeated cycles of editing, assembly and simulation in a manner similar to the debugging of software.

An interconnection simulation routine will be used to debug configurations of modules. This routine will be an event-table simulator which enables the system designer to observe the interaction of the modules in a system. The degree of overlapped operation can be observed and the effects of alterations to individual modules on the configuration can be ascertained.

When ROM designs are completed, each object program can be dumped to paper tape. A ROM can then be physically constructed by "burning" the pattern specified on the paper tape into PROM integrated circuits.

System implementation would be accomplished by the software simulation process of ROM design, followed by ROM pattern fabrication. The ROMs would then be plugged into the appropriate micromodules obtained from stock. Standard cables, also obtained from stock, would be used to interconnect the modules.

#### V. Networks of Modules

An important goal of the Micromodules project is to facilitate the construction of more complex equipments than are feasible with traditional methods of constructing hardware. In particular, we wish to encourage the use of large networks of modules. Two adaptations of well known techniques are expected to be of general use in such systems: pipelining and parallel processing.

##### A. Pipelines

Pipeline structures are particularly appropriate to the micromodules. Let each packet of data be represented as  $x$ . Let the function  $f(x)$  be computed by a pipeline of  $n$  stages, thus

$$f(x) = f_1(\dots f_{n-1}(f_n(x))\dots)$$

as illustrated in figure 1.

In designing a pipeline, each processing element should compute the appropriate function in a fixed time period. Thus each packet spends an identical amount of time in each processor. If the subfunctions to be computed do not have identical computation times, synchronization circuitry must be included in the design. If some of the subfunctions require random computation times, the buffering of data packets must also be provided for the sake of efficiency. Furthermore, each processor should be designed so that its average computation time is approximately equal to that of the other processors. Because of these optimization problems, pipeline structures are not often practicable for the implementation of complex functions.

However, a modified version of the structure (figure 2) allows the pipeline concept to be applied to a larger class of practical problems. Let the packet  $y$  be defined as the augmented pair of elements

$$y = (i, x)$$

where  $i$  is a tag value (initially,  $i=n$ ) representing the next subfunction to be computed, i.e.  $f_i(x)$ . Let there be  $m$  processing elements  $g_j$ , for  $j=1, \dots, m$ . After each processing element, there is a buffer  $q_j$  of fixed size. Let  $b_j$  be a Boolean feedback signal from  $q_j$  to  $g_j$  such that

$$b_j = \begin{cases} 1 & \text{iff } q_j \text{ is more than half full} \\ 0 & \text{otherwise.} \end{cases}$$

The  $b_j$  signal allows the processing element to determine the state of its output buffer. By considering the tag value  $i$  of its current packet and the state  $b_j$  of the output buffer, each processor makes the decision to pass the current packet to the buffer or to compute the next subfunction. Thus

$$g_j = \begin{cases} g_j(i-1, f_i(x)) & \text{iff } (b_j=1) \text{ and } (i>0) \\ (i, x) & \text{otherwise} \end{cases}$$

$$\text{for } (0 < j < m+1).$$

Note that  $b_m=1$  regardless of the state of  $q_m$ . This fact allows each processing element to contain the same microprogram. Furthermore, the microprogram is not dependent on  $m$  or  $n$ . Thus a pipeline executive microprogram can be written and debugged for arbitrary  $m$  and  $n$  values. The executive would only be concerned with reading and writing data packets, and with making the decision to process or pass the current packet. System design of a pipeline could be performed by combining the executive with a list of packet sizes and subfunction addresses in a table indexed by  $i$ , and the microcode for each subfunction.

Since the pipeline structure does not depend on  $m$ , fast failure recovery is facilitated. The faulty module can be quickly removed from the pipeline and the system can be restarted with a structure of size  $m-1$ . Performance would be degraded, but the structure could still operate with up to  $m-1$  failures.

It was stated previously in this paper that loop-free interconnect structures could sometimes be implemented for algorithms which contain loops. The simplest method would be to contain the loop within a single module by means of the microprogram. Loops can also be integrated into the pipeline structure by a modification of the definition of  $g_j$ ; let the subfunction microcode also compute  $s_i(i)$ , the next value of the tag  $i$ . Thus

$$g_j = \begin{cases} g_j(s_i(i), f_i(x)) & \text{iff } (b_j=1) \text{ and } (i>0) \\ (i, x) & \text{otherwise} \end{cases}$$

with each iteration of a loop being considered as a new invocation of the same subfunction.

Either definition of  $g_j$  preserves the order of packet throughput. Although one packet can be completely processed in the first element and another packet partially processed by each stage, each packet will leave  $q_m$  completely processed and in the original order.

Because of the importance of the pipeline concept, a data flow simulator has been programmed. The system designer can specify

the type and shape of computation time distributions (assuming they are independent) and the packet size for each value of  $i$ . By selecting values of  $m$  and by combining or reducing subfunction definitions, he can determine the most effective implementation of those considered.

## B. Parallel Processing

The use of parallel structures like that in figure 3 is also anticipated. An input controller  $I$  is used to schedule the flow of input data to each of the  $m$  processors  $g_j$ , while output controller  $O$  merges the result streams. The mode of operation depends upon the characteristics of the function  $f$ . If  $f$  requires a nearly constant amount of processing time regardless of the data packet values, a phased sequence of processing can be scheduled by  $I$  and  $O$ . If packet transfer time is  $t_d$  and computation requires time  $t_f$  for each packet, then for maximum throughput,

$$m > \lceil (t_f + 2t_d) / t_d \rceil \quad \text{for } t_d > 0$$

If  $t_f$  is random with a significant variation, a more complex structure and scheduling algorithm might be used. Data packets can be buffered as shown in figure 4 with the scheduling controlled by the state of the buffers. For the input controller, let  $b_j$  be the Boolean state signal defined previously. Then a good scheduling algorithm might be

$$n = \min(j) \text{ such that } b_j=0$$

where  $n$ , if defined, is the subscript of the next buffer  $q$  to receive a packet of data. Similarly, if  $a_j$  is the Boolean state signal for the  $j$ th output buffer, then let

$$\begin{aligned} k &= \min(j) \text{ such that } a_j=1 \\ \text{else if no } a_j=1, \\ k &= \min(j) \text{ such that } r_j \text{ is not empty.} \end{aligned}$$

For maximum throughput,

$$m > \lceil E((t_f + 2t_d) / t_d) \rceil$$

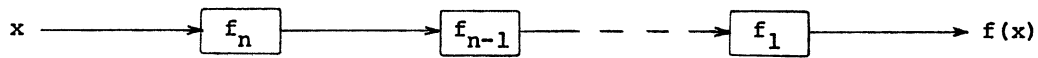
Note that the order of packet input is not preserved at the output for the case of random scheduling. If order must be preserved, then an order index can be attached to each packet at  $I$ . This index can be used by  $O$  to place the results in order. Let  $l$  be the maximum number of packets containable by the system in figure 4. Let  $u$  be the maximum possible computation time, and let  $v$  be the minimum. Then three buffers of size  $w$  are required for reordering where

$$w = \lceil (lu)/v \rceil$$

For the random scheduler, a data flow simulation is planned that will be similar to that for the pipeline structure. Several executive routines for phased and random schedulers, with and without order indexing, will be written.

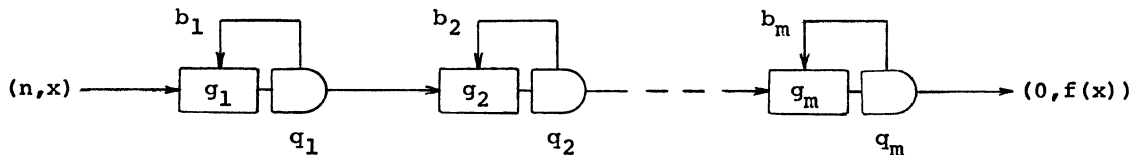
## VI. Summary

The Micromodules project is directed towards the simplification of hardware design and implementation. A powerful and flexible



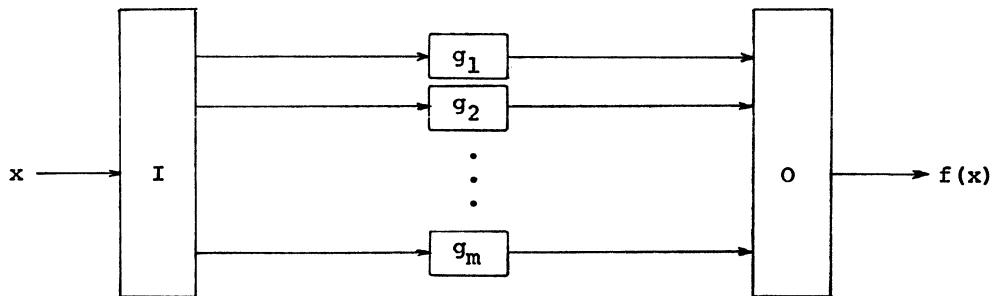
$$f(x) = f_1(\dots f_{n-1}(f_n(x))\dots)$$

Figure 1. A Pipeline Structure



$$g_j = \begin{cases} g_j(i-1, f_i(x)) & \text{iff } (b_j=1) \text{ and } (i>0) \\ (i, x) & \text{otherwise} \end{cases}$$

Figure 2. A Self-Optimized Pipeline Structure



$$g_j(x) = f(x) \quad (0 < j < m+1)$$

Figure 3. A Parallel Processing Structure

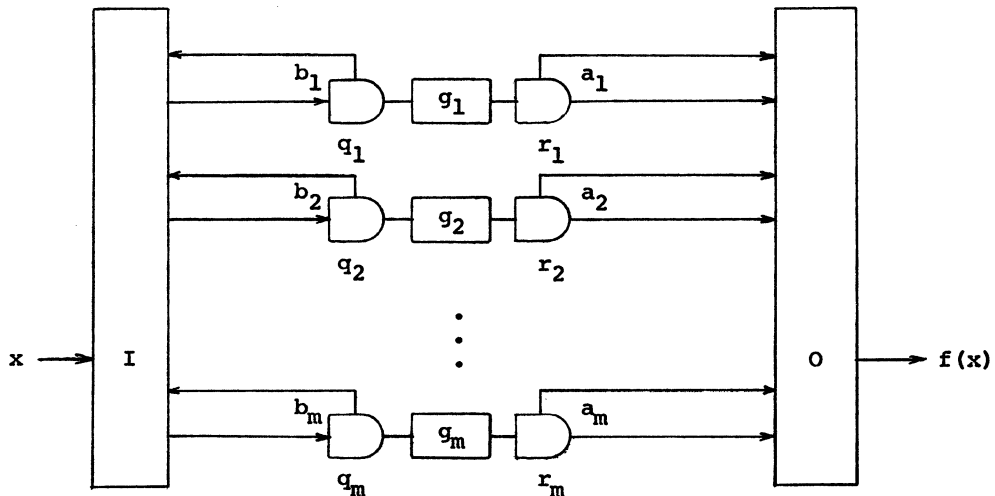


Figure 4. A Self-Optimized Parallel Processing Structure

set of microprogrammed modules is provided. The use of a standardized interconnection discipline, with an emphasis on the elimination of electrical errors, allows the engineer to concentrate on the architectural aspects of his problem.

The system designer will have two powerful structures at hand: the pipeline and parallel schedulers. He can design microprograms for the functions to be computed. Using the computation time characteristics of the function microprograms, he can simulate a data flow model and manipulate the model to achieve the desired throughput. Finally, he can assemble an arbitrary network of modules without bearing the burden of synchronization and buffering design.

A basic family of four micromodules is now in the development stage. Future work will include the identification and realization of other useful structures, whether microprogrammed or hardwired. A continuing effort to construct ROM designs with broad applicability and to further improve design aids is anticipated. Some effort will also be made to discover other basic system structures which would be useful in distributing processing tasks among a collection of modules.

#### References

- [1] W. A. Clark, "Macromodular Computer Systems", 1967 SJCC Proceedings, p335
- [2] S. M. Ornstein, M. J. Stucki and W. A. Clark, "A Functional Description of Macromodules", 1967 SJCC Proceedings, p337
- [3] C. E. Molnar, S. M. Ornstein and A. Anne, "The CHASM: A Macromodular Computer for Analyzing Neuron Models", 1967 SJCC Proceedings, p393

#### Bibliography

- [4] H. H. Loomis, Jr. and M. R. McCoy, "A Scheme for Synchronizing High Speed Logic: Part I." and "... Part II.", IEEE Trans. Computers, January and February 1970
- [5] H. H. Loomis, Jr., "The Maximum Rate Accumulator", IEEE Trans. Computers, August 1966
- [6] C. G. Bell and J. Grason, "Register Transfer Modules (RTM) and their Design", Computer Design, May 1971
- [7] D. Misunas, "Petri Nets and Speed Independent Design", Comm. ACM, August 1973

# COMPUTER MODULES: AN ARCHITECTURE FOR LARGE DIGITAL MODULES\*

S. H. Fuller  
D. P. Siewiorek  
R. J. Swan

*Departments of Computer Science and Electrical Engineering  
Carnegie-Mellon University*

## ABSTRACT

This paper describes the architecture of Computer Modules, or CMs. They are large digital modules of about minicomputer complexity that are specifically designed to take advantage of the rapidly advancing semiconductor technology. These modules are intended to be interconnected into systems that implement a wide range of computational structures. The main features of a CM include a small processor as the primary control element and memory distributed among the CMs in the system rather than centralized into memory modules as in current multiprocessors. CMs are interconnected into a network via buses that each have their own virtual address space to facilitate efficient inter-module memory sharing. This paper includes an ISP description of the address translation mechanisms as well as a discussion of several important implementation issues such as the avoidance of deadlocks in CM networks and the width of the inter-CM buses.

## 1. INTRODUCTION

This paper describes a set of digital modules that is being developed to exploit continuing advances in semiconductor technology and to enable the construction of high performance computer structures. These large digital modules, called Computer Modules or simply CMs, were introduced in two earlier papers [3,10]. These papers described some of the fundamental ideas that form the basis for our current research. Here we take a more detailed look at Computer Modules. Their architecture, as well as some important implementation issues, are discussed in depth.

In the past 15 years, standard module sets have evolved from circuit elements, to gates and flip-flops, and to register-transfer level modules (i.e., MSI packages) [11]. Continuing advances in semiconductor technology led us to enlarge the scope of our earlier work in register-transfer level modules [4] to include the study of "larger" digital modules that can exploit the emerging LSI components. Although Computer Modules originated in an effort to "scale up" the results of work on register-transfer modules, we have also learned much from "scaling down" some of the principles of multiprocessor systems currently under development [1, 13, 16]. While Computer Modules can be used to implement a general purpose computation facility, they are primarily intended for special purpose systems. The topology of the network will reflect the interprocessor communication requirements of the application.

The following description of Computer Modules is divided into two main sections: their architecture and their implementation. The architecture centers around the three types of address spaces used in CM networks and the features that enable a CM network to achieve a "tighter-coupling" between processors and memory than is possible with other multiprocessor or multicomputer organizations. Section 3 focuses on the more important implementation issues and presents specific solutions for CMs based on these considerations.

## 2. THE ARCHITECTURE OF COMPUTER MODULES

### 2.1 Fundamental Characteristics of Computer Modules

The primary control element of a Computer Module is a programable processor. This processor may be microprogramable to allow tuning to particular tasks. Microprogramable processors are versatile control elements and are currently used in such diverse systems as general purpose processors, I/O processors and controllers, device controllers and special purpose language processors. The use of small (microprogrammed or otherwise) processors as a primitive component is rapidly becoming economically feasible. Already a number of small processors exist in a single LSI package or a small number of packages [11]. This is in contrast with systems built with MSI components which must revert to small-scale integration, i.e., gates and flip-flops, to implement control primitives. Several attempts have been made to develop a control element for register transfer level modules [4, 5, 15], but they lack general acceptance. Primarily, this is because they have not been produced by any semiconductor manufacturer as an MSI component.

While there certainly exist many applications where a single, small processor is sufficient, it is clear that one processor cannot provide enough computational power to implement the range of high performance computing systems that are needed. If we hope to exploit LSI technology, some effective means must be found to interconnect a number of small processors into a network. The inter-module communications mechanisms must provide for fast communication with a maximum potential for concurrency. In Sec. 2.2 we describe in detail the scheme for interconnecting CMs.

The overall structure of a Computer Module is shown in Fig. 2.1\*. It consists of a processor,  $P_c$ ; a local memory,  $M_p$ ; a number of ports,  $K$ , maps, which allow interconnection to other CMs; and an intra-CM switch,  $S$  (processor, bus, and memory) or simply  $S.pbm$ ,

\*This research is supported by National Science Foundation Grant GJ 32758X.

\*The PMS notation of Bell and Newell [2] is used throughout this discussion to describe the organization of Computer Modules.

which allows the Pc or any port to communicate with the Mp or any other port.

It is useful to contrast the "classical" multiprocessor structure, in which an array of processors have access to a large homogeneous shared memory via a switching mechanism, with a CM network (see Fig. 2.2). The classical multiprocessor memory is homogeneous in the sense that memory access time is uniform for any word within a processor's address space. All memory accesses incur the delay of a single level of switching. In a CM network the memory is structured to an arbitrary number of levels. Access time to the local memory of a CM by the processor incurs effectively no switching delay. A processor accessing memory in another CM on a common inter-CM bus incurs two levels (two Kmaps, see section 2.2) of switching delay. Accessing memory via an intermediate CM incurs four levels of switching delay, etc. (In a Computer Module network and some other multiprocessor systems, a processor need not wait for a write operation to be completed before proceeding with the next memory reference. Thus, for isolated references or when there is little contention for inter-CM buses, write operations to remote memory will appear no slower than write operations to local memory.)

The nonhomogeneity of memory access time allows program locality to be used to advantage. CMs are primarily intended for special purpose applications, where a process can be bound to a processor and little or no multiprogramming is necessary. The frequently used code and data for a process is placed in the local memory of the CM which will execute it [13]. If code and data are common to several processors, but infrequently used, only one copy of each need exist. Other CMs on the same or adjacent inter-CM buses can access common items via the bus network when necessary. Thus, references within the primary locality of a program will be honored faster than with a classical multiprocessor with homogeneous memory. References outside the major locality will be honored slower than with a classical multiprocessor. In general purpose computer systems, cache memory schemes can be used to exploit dynamically detected program locality. This is effective for read only code and data, but severe difficulties arise in multiprocessor systems if shared writable data is stored in a cache as two of more possibly different copies of the same data can be created.

The switching mechanism, that provides fast shared-access to memory, may be a substantial proportion of the total cost of a classical multiprocessor system. This is the case with Carnegie-Mellon's C.mmp [16] where up to 16 concurrent accesses by 16 processors to 16 memory units are possible. Sharing N memories among N processors via a crosspoint switch, with a potential concurrency of N, requires on the order of  $N^2$  switching elements. If a distributed switch is used, e.g. as in the B.B.N. multiprocessor [13], on the order of  $N^2$  cables are also required. Inter-CM memory accesses will be relatively infrequent if advantage is taken of program locality. Hence the full concurrency provided in a classical multiprocessor is unnecessary. Sharing N memories with N processors requires on the order of N switching elements when using a single bus.

It is also useful to contrast CM networks with typical computer networks. In a computer network, a processor is tightly coupled to its own memory and normally does not have direct access to the memory of any other computer in the network. Processes, running on different computers, communicate by exchanging messages which are routed by some combination of hardware and software. Messages are usually long relative to the word size of the computers. In the UCI Distributed Computer System [9], messages contain approximately 1000 bits. In CM networks, communication between processes can occur at the single word level and all routing is done by hardware over high performance buses. In computer networks, inter-process

communication usually occurs only at a high level because a relatively long message must be assembled and then transmitted to a potentially geographically distant computer via relatively slow communication lines.

In summary, in a classical multiprocessor system all processors are uniformly and tightly coupled to all memory. Processes can communicate on a word level. In a CM network processors are very tightly coupled to local memory and more loosely coupled to the local memory of other CMs. Processes can communicate on a word level with slightly more overhead (minimum of two switching levels) than in a multiprocessor. In a computer network processors are very tightly coupled to their own memory and very loosely coupled to other memory in the network. Processes communicate at a message level with relatively large delays.

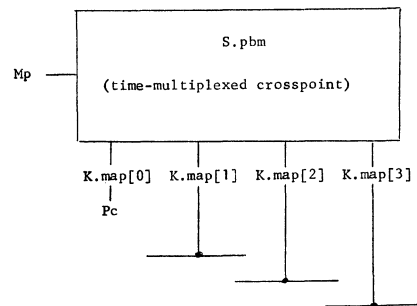


Figure 2.1 The Basic Structure of a Computer Module.

## 2.2 The Processor, Bus, and Memory Address Spaces

The inter-CM communication is based on mappings between address spaces. The three types of address space in a CM network are described in this section. To aid our discussion, Fig. 2.2 depicts a small, but non-trivial CM network: CM[A], CM[B], CM[C] and CM[D] are interconnected via inter-CM buses L and M.

The most obvious scheme to allow processors to share data and procedures in memory is to give them all the same global, linear address space. This naive scheme is lacking on a number of counts. The linear address space would have to be very large ( $2^{24}$  to  $2^{32}$ ) in order to handle many large, contemporary problems. Hence 32 bit addresses would have to be used throughout the network and the structure of the CM network would have to be coded into the memory access routing mechanisms. Instead, Computer Modules use a segmented address scheme [7, 14]. The processor sees a virtual address space, the **processor address space**, which is divided into a number of variable sized segments (currently there are 16 segments). The **memory address space** is simply the linear, physical address space of the local memory. In a single CM system the processor address space and the memory address space correspond to the virtual and physical address spaces of standard virtual memory computer systems.

Each inter-CM bus has a virtual address space. These **bus address spaces** allow processors to access the memory of other Computer Modules. For example, consider a memory reference by the processor of CM[A] to the memory of CM[B]. Figure 2.2 illustrates the mapping between address spaces that must be done:  $K.map[A][0]$  ( $K.map[0]$  of CM[A]) translates an address in the processor address space of CM[A] into the bus address space of inter-CM bus L. Now  $K.map[B][1]$  recognizes the address on bus L and translates the address into the memory address space



of CM[B]. Hence  $K.map[A][0]$  and  $K.map[B][1]$  have been used to map memory requests from the processor of CM[A] to the memory of CM[B]. The establishment of this addressability is discussed in detail in the following section.

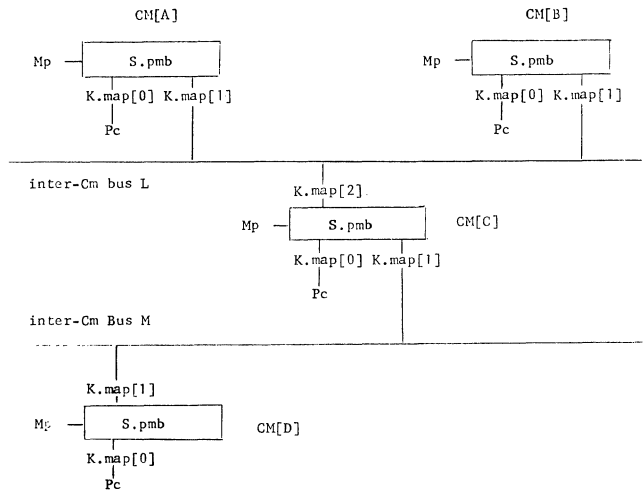


Figure 2.2 A system of four Computer Modules.

### 2.3 Routing Requests between Address Spaces

The  $K.maps$  of Fig. 2.1 each contain a segment table which specifies the mappings to be performed from one address space to another. When a  $K.map$  recognizes an address on the inter-CM bus it performs an address translation. The switch,  $S.pmb$ , routes translated addresses to either the local memory or to one of the three  $K.maps$  connected to inter-CM buses. The routing is specified for each segment in the segment tables. When a  $K.map$  receives an address from the switch it requests the inter-CM bus. The address is subsequently placed on the bus without further translation.

To return to our example of Fig. 2.2, the processor of CM[A] is able to write a word in the Mp of CM[B] if the segment tables in  $K.map[A,0]$  and  $K.map[B][1]$  are set correctly.  $K.map[A][1]$  performs bus arbitration but does not translate the address. This ability of  $K.maps$  to route single word memory access requests, independent of the processors is an important aspect of the CM architecture. It ensures a more closely coupled structure than is possible with computer networks that transfer data under the control of a communication or message processor.

Figure 2.3 illustrates another important property of CMs, their use as a switch between inter-CM buses. In Fig. 2.3 the appropriate address spaces are shown for  $Pc[A]$  (the processor of CM[A] in Fig. 2.2) to access a word in  $Mp[D]$  (the memory of CM[D]).  $K.map[A][0]$  maps the request from  $Pc[A]$  into the bus address space of inter-CM bus L;  $K.map[C][2]$  maps the request from inter-CM bus L into the bus address space of inter-CM bus M; and finally,  $K.map[D][1]$  maps the request into the memory address space of  $Mp[D]$ . An alternative to this scheme is to have a second module type. It is more efficient, however, to take advantage of the address translation and bus interface logic already provided within a CM than to duplicate it with a special purpose switch module. The ability to automatically route single-word memory accesses to any memory in a network is crucial if networks of CMs need to interact in a closely-coupled manner. Transfer of blocks of information is also important in

many applications. Special hardware is provided so that the processor may initiate a block transfer and then continue program execution. This allows a higher data transfer rate and more productive use of the processor than block transfers by program.

Figure 2.4 illustrates additional ways the bus address space can be used. In Fig. 2.4(a) several CMs are set up to map the same bus segment into their local memories. This arrangement gives CM systems a broadcast, or one-to-many mapping ability. For example, CM[A] in Fig. 2.4(a), writing into a single location sends information simultaneously to the Mps of CM[B] and CM[C]. On the other hand, Fig. 2.4(b) shows how a CM system can implement a many-to-one mapping. This arrangement is needed whenever several concurrent processes share a common data structure.

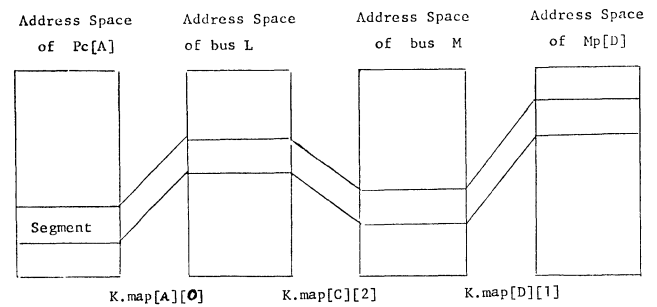
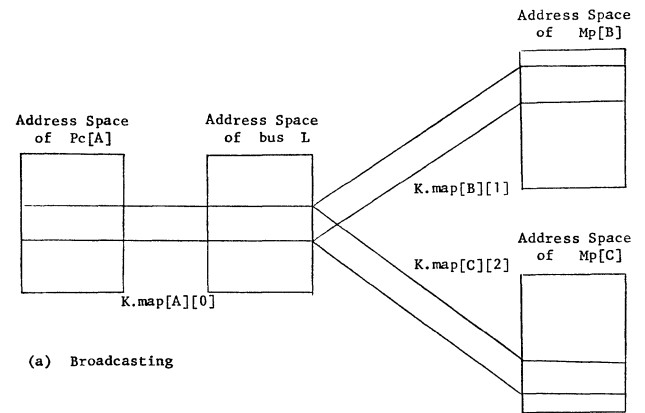
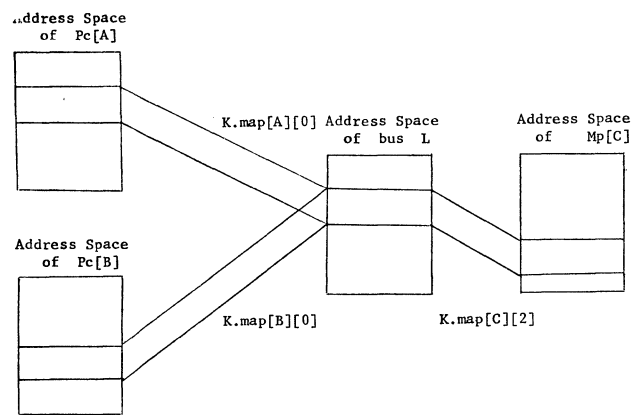


Figure 2.3 Address translation with a CM used as a switch.



(a) Broadcasting



(b) Sharing

Figure 2.4 One-to-many (a) and many-to-one (b) address mapping.

## 2.4 The Segment Table and Address Translation Mechanism

The mapping logic within the K.map actually performs two distinct functions: address recognition and address translation. Address recognition and translation information is held in the segment table of each K.map. All the access modes described in the previous section are achieved by appropriate segment table entries. This section describes in detail the address translation mechanism. It is also concisely defined, in ISP notation [2], in the appendix.

The segment table consists of a set of segment descriptors, where each descriptor defines how one segment in the source address space is mapped into a segment in the destination address space. A segment descriptor is composed of three fields: the source segment name, the destination segment name, and the control and status field. One of the subfields in the control field is the logarithm, base 2, of the segment size. Hence, only segment sizes that are powers of two are allowed: 1, 2, 4, 8, ..., 4K. By restricting the segment sizes in this way we avoid the addition implicit in more conventional base/limit translation schemes.

Figure 2.5 shows the essential properties of the mapping function performed by the K.map mapping logic. The four most significant bits of the source address are used as an index into the segment table to select a row, or segment descriptor. The segment size field, of the selected descriptor, specifies the position of the boundary between the segment name and the displacement fields in the source address. The segment name field of the source address is compared with the segment name in the descriptor. If they match, the destination address is generated by concatenating the destination segment name with the displacement field of the source address. This mechanism ensures unique recognition of segments provided that each segment is allocated at a base address that is divisible by its size (which is a power of 2).

For a write operation it is reasonable that two or more K.maps, on the same inter-CM bus, respond to the same address (c.f. section 2.3 and Fig. 2.4 (a)). For read operations to the same address it is necessary to ensure that only a single K.map responds. Thus, apart from conventional protection issues, it is necessary to be able to specify that a segment is **read protected**. It is also possible to entirely disable a segment descriptor, i.e. prevent it from ever matching.

All the segment descriptors are in the local memory address space. This circumvents the need for the processor to have special instructions to maintain the segment descriptors and also provides a clean protection mechanism for the descriptors. By suitable setting of the segment descriptors they can be made available to a remote CM for access via one or more inter-CM buses. The local PC can relinquish its ability to access its own segment descriptors. Thus centralized control of inter-CM communication is possible and faulty PCs can be effectively removed from a network.

## 2.5 Coordination of Control Between CMs

Inter-CM coordination is a direct extension of the memory sharing mechanism described in the previous section. Any segment, which maps to the local physical memory address space of the CM, may be specified as a control segment. An attempt to write into a control segment causes an interrupt to the processor and control is transferred to that effective address in the processor address space. The contents of memory is not altered. However, the data part of the write operation is saved and can be used as a parameter to the interrupt routine. It could contain the identity of the interrupting CM, the value of a parameter, a pointer to a list of parameters, etc. In multiprocessor systems it

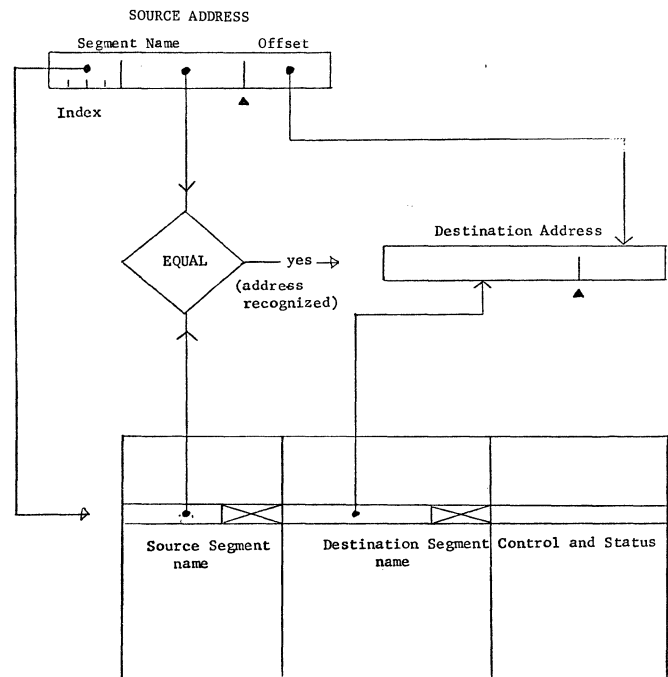


Table of 16 Segment Descriptors

Figure 2.5 Address recognition and translation by a K.map.

is usually necessary to provide synchronization primitives, e.g., the P and V operators of Dijkstra [8]. CMs allow the implementation of these synchronization primitives.

## 2.6 CMs as Modules.

There are many aspects to the design of digital modules: interconnection rules, number of external pins, individual performance, etc. Here we are concerned with the extensibility of a network of Computer Modules. In this context, extensibility applies to several dimensions: address space, total memory, total processing power and total data transfer rate. The address space of an individual processor is, of course, limited by its internal address size. The address space can be expanded by reloading the segment registers that provide addressability to the local memory of other CMs via the inter-CM buses. Further addressability can be achieved by altering the segment registers of intermediate CMs used as switches. Thus CMs can be arranged in tree structures to give addressability to an arbitrarily large memory (with potentially considerable delays).

Memory can be added to a CM network both by increasing the local memory of each CM and by adding extra CMs. Similarly, an arbitrary number of CMs can be added to give extra processing power. Extra processing power in this form is useful only if the task can be partitioned to execute in parallel on the extra processors.

The most direct method of adding CMs to a network is to extend an existing bus. There is no limit, at least conceptually, to the number of CMs per bus. However, since the maximum data transfer rate per bus is fixed, there is a limit to the useful number of CMs per bus. It is usually possible to increase the overall data transfer rate on a bus (with more than three communicating CMs) by grouping the CMs by their frequency of interaction, dividing the bus in two, and inserting a CM as a switch.

### 3 MAJOR IMPLEMENTATION ISSUES

#### 3.1 Address Translation Logic and the Intra-CM switch.

It is important to find an efficient and economic implementation of the ports and the internal switch of a CM, since their complexity may exceed that of the processor. The intra-CM switch, S.pbm, could be implemented as a cross-point switch to provide maximum concurrency. However, consideration of locality indicates that a large majority of the traffic through the switch will be from the processor to local memory. Less traffic will pass between the Pc and the inter-CM buses and from the inter-CM buses to memory. Normally only a small fraction of traffic will pass from one inter-CM bus to another. This suggests that there would be little performance loss if the switch had a concurrency of one. By the same argument, much of the address translation logic within each K.map can be centralized into a single shared unit. Full centralization of the address mapping logic may degrade the performance of the inter-CM buses by increasing the effective address recognition/rejection time. Partial centralization, however, provides significant hardware savings while minimizing the effect on performance. K.map[0] may be treated as a special case with accesses to local memory by the processor being treated in a simpler, faster manner than accesses to remote memory.

#### 3.2 Deadlock with Inter-CM Memory Access

In a CM network it is clearly necessary to ensure that deadlock does not occur with inter-CM memory access. A set of processes is defined to be deadlocked[12] when no process can proceed without acquiring a resource already held by another process within that set. The necessary conditions for deadlock are: resources must not be sharable or pre-emptable, resources must be retained while a process is acquiring further resources, and there must be a circularity in the resource requirements of the processes.

Referring to the four CM network of Fig. 2.2, consider mutual memory accesses between CM[A] and CM[D] where CM[C] is used as a switch and may be executing a program independent of the communication between CM[A] and CM[D]. An address generated by the processor Pc[A], which is intended to reference the local memory of CM[D], will be translated by K.map[A][0] and passed to K.map[A][1]. When the K.map[A][1] has acquired control of inter-CM bus L the address will be placed on it and then recognized and translated by K.map[C][2]. K.map[C][1] will acquire control of inter-CM bus M and the address will be placed on it. This address will be recognized and translated (for the third time) by K.map[D][1], and is now used to access a word in Mp[D]. This method of implementing inter-CM memory access where inter-CM buses are acquired in sequence and relinquished in reverse order we call circuit switched.

Consider the consequences of concurrent mutual memory requests by CM[A] and CM[D] in Fig. 2.2. It is clear that a situation may arise where Pc[A] holds bus L and Pc[D] holds bus M. Unless one of the processors can be forced to relinquish a bus, neither memory access can be completed and the network is deadlocked. In this simple network the impasse will be clearly evident at CM[C] and one request may be chosen arbitrarily for pre-emption, thus resolving the deadlock. Deadlock can occur, even with a trivial two CM network, if they are implemented with an internal bus which is common to the processor and the local memory, e.g. the PDP-11 Unibus[6]. To avoid deadlock it is essential that a processor be able to make an external reference while its own local memory is being referenced.

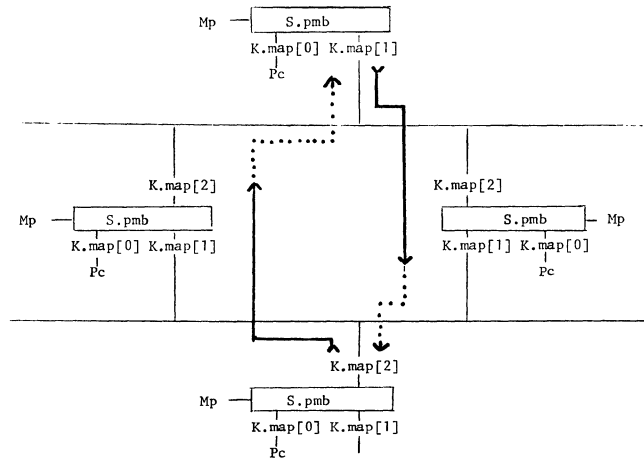


Figure 3.1 Deadlock, indistinguishable from congestion, with circuit switching.

In networks where there is more than one possible access path between any two buses, deadlock may occur without any single K.map or CM being able to detect it. Figure 3.1 shows a deadlock situation which is manifest at two K.maps in the network. Without global information neither of the K.maps can distinguish the situation shown from a condition of simple congestion. A timeout mechanism could be used after which an incomplete access attempt is pre-empted and later retried. This mechanism would be very inefficient. There would remain a possibility of recurrent deadlocks by conflicting access requests since there is no way to avoid the conflicting access requests being pre-empted approximately simultaneously. With circuit switching, in an arbitrary network, the only way to guarantee freedom from deadlock is for each request to carry a unique priority. This would ensure that one request is able to complete when an impasse occurs.

If memory access between computer modules can be implemented without a request holding more than a single bus at any time then deadlock over the allocation of buses can be eliminated. This requires that information which defines the memory request be buffered at each CM or K.map on the access path. For read operations it also requires that the buses which comprise the access path be reacquired\* to propagate the data back to the requesting processor. This type of inter-CM memory access implementation we call element switching. An element is the information that defines a memory access request (address, control signals and usually data). An element is analogous to a message in a computer network but is considerably shorter.

Although element switching eliminates deadlock with respect to the allocation of buses it introduces the possibility of deadlock over the allocation of element buffers. Provision of one buffer per access path through a CM is sufficient to guarantee freedom from deadlock. (This property, and other aspects of the deadlock situation, are the subject of continuing investigations.) Element buffers are allocated by associating a distinct buffer (approximately 40 bits) with each segment mapped by a K.map. This buffer allocation mechanism is sufficient to enable all possible access networks to be implemented without deadlock provided sufficient segment descriptors and/or CMs are available.

\*The read element marks the access path on the forward journey. This enables the address used in the forward direction to determine a unique return path to carry the data referenced to the requesting processor.

Element switching provides better bus utilization and hence alleviates inter-CM bus contention. If there is no bus contention, element switching will increase the time overhead in making a read access to a remote memory over a corresponding circuit switched implementation. The extra time overhead is incurred in reacquiring the buses to deliver the data to the requesting processor.

### 3.3 The Width and Nature of the Inter-CM Bus

The number of external pins required to interconnect Computer Modules will have a significant impact on their cost. A related factor is the amount of heat dissipated when driving an external line. Power dissipation may be the dominant factor limiting packing density for an LSI implementation of CMs.

While reliability and economic considerations lead to narrow buses and few pins, performance considerations clearly imply that the inter-CM buses have a high data bandwidth which is facilitated by wide buses. Both the overall potential maximum data transfer rate and the response time for individual read requests (assuming no contention) are important measures of the inter-CM bus performance.

Minicomputer system buses usually have distinct lines for each function (address, data and control) and are fully interlocked on a word-by-word basis. The absence of time-division multiplexing of the information carrying lines allows for a high data-transfer rate and a minimum of complexity in devices interfaced to the bus. Interlocking provides reliable operation over a range of bus lengths and loading conditions. Analysis of bus handshaking protocols shows that time multiplexing of the information lines between address and data imposes considerably less than the two to one time overhead expected. Half-width buses retain the inherent reliability of full interlocking while significantly reducing the number of pins and cables required for bus connections.

Further reduction in the number of lines per bus may be achieved by increased time multiplexing of the information lines. To maintain data transfer rates comparable with a full or half-width bus the full interlocking on each bit must be sacrificed. We are investigating schemes which employ a total of 4 to 10 lines per bus. Address and data information is transferred as self clocking pulse trains down each line. Bus control functions are performed using the same lines that carry information.

## 4. CONCLUDING COMMENTS

**Parallel Algorithms.** Computer Modules are intended to facilitate the implementation of parallel algorithms. However a general solution to the problem of decomposing a task into efficient parallel processes is not near at hand. Nevertheless, there exist parallel algorithms for some important problems and there are many applications where the task is presented in a decomposed form. For instance, most process control applications consist of a number of specialized control tasks with communication at a higher level occurring infrequently.

**Project Status.** Currently research is proceeding on two fronts. A detailed simulation of a CM network is being written. Particular emphasis is being placed on the effect of interactions between external memory accesses so that the effects of bus and memory contention can be accurately assessed. A number of applications will be run on the simulation with a wide range of CM configurations. Concurrent with the development of the simulation a small number of CMs will be built based on existing MSI components and commercially available processors. These will

provide performance figures for the simulation and demonstrate the technical feasibility of the design.

## REFERENCES

- [1] H. B. Baskin, et al, "A Modular Computer Sharing System," *CACM*, Vol 12, No. 10, October 1969.
- [2] C. G. Bell, and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York (1971).
- [3] C. G. Bell, R. C. Chen, S. H. Fuller, J. Grason, S. Rege, and D. P. Siewiorek, "The Architecture and Applications of Computer Modules: A Set of Components for Digital Design," *IEEE CompCon '73*, (March 1973), pp 177-180.
- [4] C. G. Bell, J. L. Eggert, J. Grason and P. Williams, "The description and Use of Register Transfer Modules (RTMs)," *IEEE Transactions on Computers*. Vol C-21, No. 5, May 1972, pp 495-500.
- [5] W. A. Clark, "Macromodular Computer Systems," *AFIPS Conference Proc.*, Vol 30, SJCC 1967, pp 335-336.
- [6] DEC "PDP-11/40 Processor Handbook," *Digital Equipment Corporation*, 1972.
- [7] P. J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2. No. 3, September 1970, pp 153-190.
- [8] E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, Genuys (ed.), Academic Press, London, 1968.
- [9] D. J. Farber, and K. C. Larson, "The System Architecture of the Distributed Computer System - The Communication System," Presented at the Polytechnic Institute of Brooklyn, Symposium on Computer Networks, April 1972.
- [10] S. H. Fuller and R. C. Chen, "The I/O port Architecture for Computer Modules," Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, Pittsburgh Pa. 15213. (March 1973).
- [11] S. H. Fuller and D. P. Siewiorek, "Some Observations on Semiconductor Technology and the Architecture of Large Digital Modules," *IEEE Computer*, Vol. 6, No. 10, October 1973, pp 14-21.
- [12] A. N. Haberman, "Prevention of System Deadlocks," *CACM*, Vol. 12, No. 7, July 1968.
- [13] F. E. Heart, S. M. Ornstein, R. W. Crowther and W. B. Barker, "A New Minicomputer/Multiprocessor for the ARPA Network," *Proc. AFIPS NCC 42*, 1973, pp 529-537.
- [14] B. Randell, and C. J. Kuechner, "Dynamic Storage Allocation," *CACM* Vol. 11, No. 5, May 1968, pp 297-305.
- [15] D. M. Robinson, "Digital System Design with Control Modules," *IEEE CompCon '73*, March 1973, pp 207-210.
- [16] W. A. Wulf and C. G. Bell, "C.mmp - A Multi-Mini-Processor," *AFIPS Conf. Proc.* Vol 41, part II, FJCC 1972, pp 765-777.

*Appendix: ISP Description\* of the K.map Address Translation*

K.map State

Segment\_Descriptor\SD[0:15]<41:0>  
*The 16 descriptors in the segment table define the mapping between the source address space and the four destination address spaces.*

Source\_Segment\_Name\SSN                    := SD<41:30>  
*Name of source segment, 4 high order bits of name are given by the position in the table.*

Destination\_Segment\_Name\DSN               := SD<29:14>  
*Name of destination segment.*

Control\_and\_Status\_Field\CSF<13:0>       := SD<13:0>  
 Log\_Segment\_Size\LSS<3:0>                := CSF<13:10>  
*Size of segment in both source and destination address spaces is 2<sup>LSS</sup>. LSS ≤ 12.*

Mask<11:0>                                 := LSS ↓ 1  
*The unary encoding of LSS.*

Destination\_Address\_Space\DAS<1:0>       := CSF<9:8>  
*Designates the address space of the result: memory or the three inter-Cm bus address spaces.*

Descriptor\_Active                           := CSF<7>  
*Set to allow translation with this segment\_descriptor.*

Write\_Protect                              := CSF<6>  
*Block translation of write requests.*

Read\_Protect                                := CSF<5>  
*Block translation of read requests, required when one-to-many address mappings are used.*

Referenced                                 := CSF<4>  
*Set whenever segment\_descriptor is used.*

Changed                                     := CSF<3>  
*Set whenever the segment descriptor is used for a write request.*

Control\_Segment                            := CSF<2>  
*Force interrupt on a write attempt to this segment, see text.*

Interrupt\_Enable                           := CSF<1>  
 Interrupt\_Pending                         := CSF<0>

Interrupt\_Data\_Buffer<15:0>  
*Register to hold parameter (data part of write operation) of an inter-module interrupt request. Only a single buffer is necessary because subsequent interrupt requests queue in the element buffers which are not visible to the programmer.*

Address Translation Process

x<15:0>  
*Address of request in source address space.*

Index<3:0>                                 := x<15:12>  
 Field<11:0>                               := x<11:0>

y[0:3]<15:0>  
*Result address in one of four destination address spaces.*

Descriptor\_Active[Index] ∧  
*Test that indexed descriptor is active.*

(Field ∧ ¬Mask[Index] = SSN[Index] ∧ ¬Mask[Index]) ∧  
*Test source address matches source segment name.*

(¬Write\_Protect[Index] ∨ {x is read request}) ∧  
 (¬Read\_Protect[Index] ∨ {x is write request})  
*Check protection.*

⇒ ((¬Control\_Segment[Index] ∨ {x is read request})  
 ⇒ y[DAS[Index]] ← (Field ∧ Mask[Index]) ∨  
 (DSN[Index] ∧ ¬Mask[Index]));  
*This is the translated address.*

((Control\_Segment[Index] ∧ {x is a write request}) ⇒  
*Test for interrupt.*  
 ( Interrupt\_Data\_Buffer ← data;  
*Save data from write request.*  
 Interrupt\_Pending ← 1 ; next  
 Interrupt\_Enable ⇒ {interrupt Pc}))

Block Transfer Mechanism State

Source\_Address<15:0>  
*Start address of source block.*

Destination\_Address<15:0>  
*Start address of destination block*

Block\_Transfercontrol\BTC

Enable\_Transfer                            := BTC<15>  
 Interrupt\_Pending                         := BTC<14>  
 Interrupt\_When\_Complete                 := BTC<13>  
 Error                                     := BTC<12>  
 Word\_Count<11:0>                         := BTC<11:0>

*The block transfer mechanism operates within the Pc address space.*

\*This ISP has been simplified for clarity.



# A MICROPROGRAMMED ARCHITECTURE FOR FRONT END PROCESSING

Rodnay Zaks

*Universite de Technologie de Compiègne, France*

## INTRODUCTION

The increasing diversity of hardware devices and software procedures developed for remote processing has yielded a multiplicity of new facilities and telecommunication network structures. The corresponding architectures for front-end systems range from specialized device control to sophisticated multi-purpose multi-terminal support. Simultaneously, the very complexity of new data transmission and processing techniques has created a need for flexible and powerful yet transparent communications processors. The functions of a front-end system will be analyzed in order to derive the concepts which will be used to establish a classification.

In a first part, telecommunications and control functions are analyzed in detail, as well as the user facilities at the functional level. From these concepts, two types of practical classifications are evolved. A global classification characterizes the front-end system from the operating system's standpoint. A local classification characterizes it as a local device, in function of its service capabilities to the user. This dual system allows a simple classification of a given device and hence a simpler comparison with others in its class.

The level of support provided by major commercial operating systems with respect to front-end systems is then examined. Their facilities and shortcomings provide the basis for front-end systems.

Finally, the architecture of a commercial microprogrammed front-end processor is presented. A modular microprogrammed architecture of this type allows efficient and economical system structuring for a wide range of teleprocessing services.

## FRONT-END SYSTEMS FOR TELEPROCESSING

### *The basic functions of a telesystem*

The telecommunications device, ranging from a simple hardwired controller to a large programmable processor, is coupled to the host processor's operating system via a software communications system. This global system performs all the telecommunications functions. It can be called the telesystem. According to the distribution of functions

from the host processor to the front-end device, it becomes possible to classify front-ends into logical categories. Such a classification makes a cost-performance analysis then easily feasible. In order to establish this classification, the functions of a telesystem are now considered.

The minimum functions are:

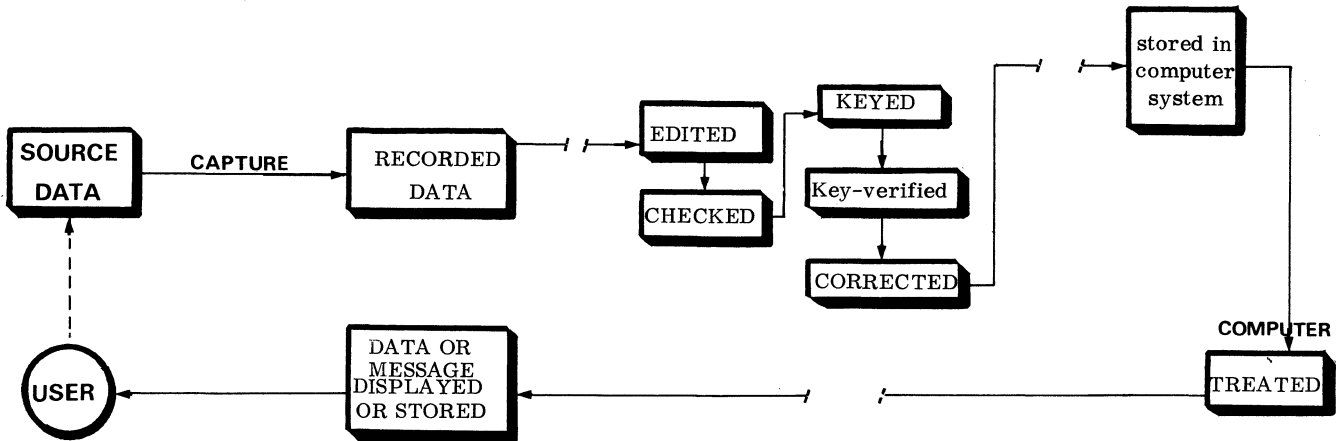
1. transmission initiation, control, and completion.
2. data assembly into required structures: from bits to words, blocks, packets, or messages.
3. code conversion according to the device, and/or host processor code.
4. error checking and recovery, possible multiple transmission, error logging.
5. recognition of control characters and special markers.
6. line monitoring.
7. message routing: to/from device or host processor.
8. bookkeeping procedures associated with beginning and end of transmission.

More complex functions which may reside at the remote station are:

1. line discipline and control: procedures for dial, polling or loop systems.
2. queuing: in a multiple-device or multiprogramming environment.
3. dynamic buffer allocation.
4. message editing and compaction.
5. local network traffic control: routing to appropriate device or process.
6. communication line concentration or multiplex.
7. priority scheduling.
8. logging.

In addition, the following specialized functions may be provided:

1. fail-soft: automatic recovery from transmission or system errors or failures
2. on-line diagnostics
3. on-line operator dialogue
4. specialized data compaction
5. input validation
6. screen regeneration
7. graphic manipulation
8. control of specialized units



**ILLUSTRATION 1 . THE STEPS OF DATA ENTRY**

9. security and access procedures
10. stand-alone capabilities.

*Distinguishing front-end configurations*

Front-end systems perform an interface role between the user and the host's operating system, or the telecommunication procedure. Each of the functions described in the preceding section can be implemented in hardware, firmware, or software. Further, most of these modules may reside either within the host system, or anywhere on the line between the operating system (or the I/O port) and the user.

This results in a variety of architectures since, once logical functions are assigned to physical or software modules, these modules may in turn appear as an arbitrarily complex front-end system. The resulting complexity does not facilitate a logical classification of such systems. Since the physical distribution of logical modules onto hardware supports may vary widely, and still accomplish similar functions, it appears practical to classify these systems by the level of service provided.

Two functional classifications will be made, characterizing the system either by its appearance to the host processor's operating system (global classification) or by its level of service to the user (local classification).

*A global classification of front-end systems*

When viewed at an operating system level, the front-end system is characterized in its global environment, and its capabilities are tied to the host operating system's capabilities that it supports or enhances. The notion of operating system becomes a global concept which may be embodied in one or more processors. It is assumed here that a significant portion of the operating system resides in the front-end.

The four essential modes are:

1. real-time.  
This applies to all cases where the front-end system can react in real-time to a modification of its environment.

It includes in particular remote process control, where the host system is viewed generally as a data base.

2. time-sharing

Where it becomes uneconomical to have a large number of slow devices interfacing directly to an I/O channel of the host processor, a front-end system may provide the desired interface functions. As in the case of real-time, this does not preclude the system from performing special functions prior to communicating with the host. Such functions may include: editing, formatting, code conversion, preprocessing, or even pre-compiling.

3. data collection

This includes all cases where the front-end system appears to the host's operating system as a collection of I/O devices, capturing and managing the data flow. This includes in particular remote-batch processing (program entry) as well as data entry, inquiry, and update. Data may be captured and collected by a variety of devices, including special-purpose devices. An essential role of the front-end system is then to look like a standard host I/O device.

4. packet switching

In this role, the front-end performs the automatic routing of blocks of information, whether messages or packets, between software or hardware modules. This implies elaborate scheduling facilities within the front-end, with queuing, dynamic buffering, and corresponding facilities within the next processor's telecommunication module.

*A local classification of front-end systems*

The front-end system is characterized in function of its specific capabilities to the user, as a local device or service.

The three main types are:

1. Emulator

The emulator is a plug-to-plug compatible device replacing an existing manufacturer's controller. Typical applications are IBM 2700 or 3700 series emulators. These may be hard-wired, micro-coded, or software-



coded devices. Hard-wired devices usually offer cost savings advantages, while firm- or soft-coded implementations allow more flexibility. This flexibility may be used for the same device to implement several emulators (communication with different host processors). It can also be used to offer extra services, in addition to the strict emulation capability. As the range of extra services increases, a second type can be characterized:

## 2. Intelligent Emulator

With diminishing hardware costs, and an ever increasing demand for varied user services, emulator devices tend to increase in sophistication and offer a new range of services, previously not available on the device they replace. An intelligent emulator may offer extended user or operating system dialogue facilities, fail-soft capabilities (message accumulation in case of host processor failures, warning to the users, orderly recovery procedures), extended terminal support, local message routing between terminals (listing cards on the local printer, and onwards to more elaborate message switching), limited data validation. A general rule is to consider as an intelligent emulator a front-end device whose basic function is emulation, and where standard functions have been extended or improved, or where service facilities have been added. As more general facilities or functions become added, a third type must be introduced:

## 3. general front-end processor

Such a system can be characterized by the fact that it still appears as a single device to the host's operating system, yet performs a range of services functionally distinct from emulation. Such a system may present advantages at two levels. At the user's level by offering services not previously available with the host manufacturer's equipment. At the system's level, by incorporating many or most of the functions previously handled by the host processor and/or specialized equipment or modules.

Typically, the front-end processor will incorporate a resident real-time operating system and handle most telecommunications functions: line polling, device dependencies, queuing, buffer management, code conversion, error recovery, multiple transmission. "Intelligent" front-ends provide in addition special software capabilities for specialized data capture and validation, field checking, applications packages, store-and-forward message switching. The linkage between the front-end processor and the host's operating system may then simply consist of a front-end control program, with nearly all telecommunication functions delegated forward. Some front-end processors even provide a stand-alone capability with high-level languages (FORTRAN, COBOL) available for local execution. This may be an attractive solution as a back-up, in case of host malfunction, or a valuable local service (night utilization).

The flexibility and power itself of a general front-end processor implies the need for operating software facilities. At a minimum, host-resident facilities should include the following functions:

- (a) front-end cross-assembler
- (b) load module, allowing to load the front-end system from its host
- (c) transfer module, allowing to dump core, or transfer

information from front-end's storage onto a host device

- (d) network configurator (macroprocessor)
- (e) file system facilities for front-end library programs.

## *Operating system support*

The various degrees of support afforded by major operating systems are examined here. The global classification of front-end systems has already introduced three main types of support: real-time, time-sharing, data collection, and message switching. Important subtypes are remote job processing, inquiry, and transaction support: access to large data files for update (write) or query (read).

Other facilities which may be included in the telecommunications module, interfacing to the Operating System, are:

### 1. message control.

It provides routing facilities between the user's program and the telecommunications facilities. In basic access, messages are simply routed to their destination point, without simultaneous multi-access. In queued access, the module manages dynamic buffers and schedules transmission.

### 2. processing module.

This user program may perform data collection and compaction, complex message switching, and on-line data file access. It may be equipped with specialized data processing packages.

## *IBM OS support*

IBM's OS is one of the most widely interfaced operating systems and deserves a special analysis. IBM 360's or 370's systems use one or more 2700 series transmission control units. S/370 may also use the 3700 series.

The 2700 is a hardwired controller with three basic capabilities:

1. character assembly/disassembly
2. control character identification
3. line monitoring (time-out inactive terminals).

All communications control is accomplished in the CPU under OS (or, in a limited way, DOS). The communications module is BTAM, QTAM or TCAM ("Telecommunications Access Method").

BTAM is a simple package which provides elementary control functions for telecommunications lines. It is invoked in the user's program by the following two macros: WRITE, to send a message; READ, to receive data. BTAM is an interface module between OS and the user program. It does not provide any queuing. Any moderately complex application then requires other telecommunications control packages as additional interfaces. These will usually reside in high priority partition (time critical I/O control).

TCAM provides the queuing facilities. It includes a traffic scheduler, handles message switching, and can support a high degree of multiprogramming. It is invoked with the GET and PUT macros.

For completeness, the channel control primitives are outlined. IBM's I/O instructions, labelled CCW (Channel

Command Words), perform three functions: data transfers, device control, branching within the channel program. Communications between the CPU and the channel are performed as follows:

1. CPU command to channel (four types):
  - (a) start I/O
  - (b) test I/O
  - (c) halt I/O
  - (d) test channel
2. channel's interrupt to CPU. The main types are:
  - (a) I/O
  - (b) programmer error
  - (c) supervisor call
  - (d) external
  - (e) machine check

Back to IBM control units, the 3700 series is a programmable processor which provides 2700 emulation or front-end facilities under NCP. In that case, it implements part of TCAM: polling, error recovery, terminal dependency, code conversion; the following functions remain in the host processor: user - OS linkage and message processing. Although the 3700 is programmable, it does not provide spectacular improvement. It is limited to 370's under OS or VS (no DOS), and does not support local peripherals. It must also be stressed that 360 OS MFT/MVT remote support is limited to remote batch initiation.

*CDC 6000 series SCOPE 3*

INTERCOM 1 provides interactive time-sharing and remote-batch processing support. It is limited to two types of terminals: teletype and CRT display. Elaborate file access and protection facilities are provided, as well as inter-user communications. In remote batch, commands issued from a terminal place a file on a batch queue for processing.

*Honeywell 200 Mod 4 OS*

The communications supervisor supports remote terminals like local peripherals. This allows remote-batch entry. In addition, query/reponse programs residing in the user's partitions provide interactive terminal communication.

*Burroughs 6500 MCP*

MCP (Master Control Program) provides elaborate communications facilities through a data communications processor. It supports remote computing, inquiring and time sharing. The message control system handles file maintenance and job control, and supplies message-switching capabilities and inter-user communications. Facilities provided include a variable number of remote stations, line monitoring, conditional command processing (detection of exception conditions), initiation of object jobs as independent processes, and maintenance of file and remote user security.

*Univac 1108 EXEC 8*

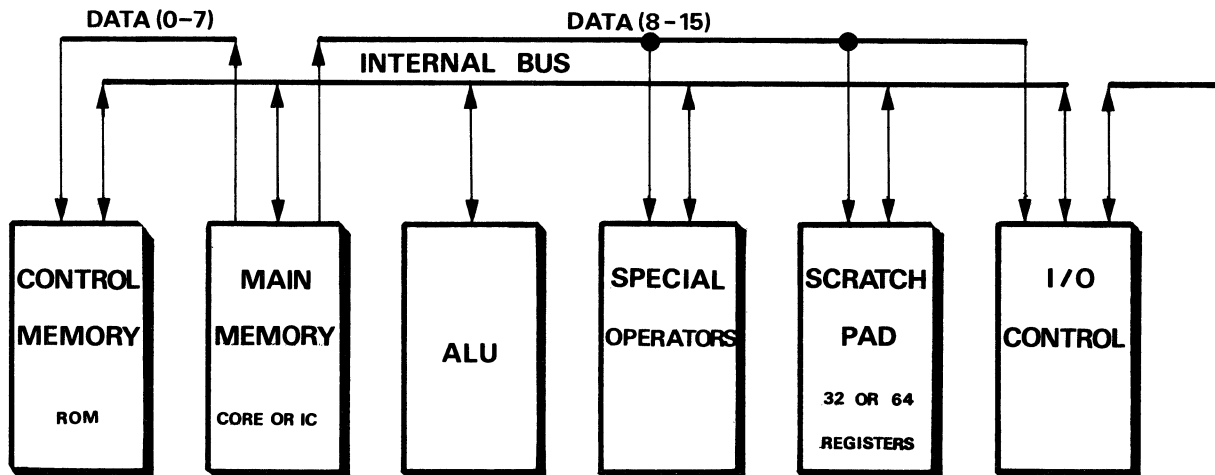
Remote facilities provide concurrent or on-demand batch and real-time processing. The executive control language allows commands to be specified from a user's remote console in conversational mode. It supports paper-tape input and generalized inter-process-communications.

*XDS SIGMA 5/7 BTM*

BTM (Batch Time Sharing Monitor) provides time-sharing access, remote batch initiation, file positioning. Only the operator may communicate with on-line users.

*Honeywell GE 600 series GECOS III*

GECOS III (GE Comprehensive Operating Supervisor) provides remote-batch and time-sharing. In batch, other



**ILLUSTRATION 2 . THE DATA FLOW**

terminals may be specified for output or messages. Direct communication with a processing program allows direct inquiries.

### A FRONT END MICROPROCESSOR

A simple 16 bit parallel microprogrammed processor system, developed for teleprocessing in Europe, is described here. In its basic version, it has been configured as a multi-procedure intelligent emulator. Its design provides an illustration of the structural trade-offs involved in obtaining the required flexibility at minimum-cost in a fairly well-defined environment. This front-end system usually interfaces directly to a host operating system via an I/O channel. In its simplest version, it is transparent to the operating system, and provides teleprocessing services such as remote-batch. Due to the large variety of teleprocessing needs, its structure will have to evolve with time. It is commercialized under the name "Ordo 16" (Société des Ordoprocresseurs).

A single internal 16-bit bidirectional bus connects all logical elements. While limiting the internal transfer speed, and reducing the possible overlap of microinstruction phases, this allows the simple insertion of specialized hardware functions as plug-in modules. A very short microword format (12 bits) limits the possible synchronicity of micro-operations, but uses a highly encoded vertical code to achieve complex arithmetic or logical operations in a single microinstruction cycle (250 nsec).

The bus connects the control memory, the main memory, the ALU (Arithmetic Logical Unit), the scratchpad (a set of 32 or 64 fast registers), and the I/O controller. This internal bus is bidirectional, half-duplex. I/O modules communicate with the I/O controller through a slower external bus

(0.666 MHz vs. 4 MHz for the internal bus). In addition, a number of special-purpose hardware modules may be inserted on the internal bus. Possible modules are: binary function generators, BCD arithmetic or conversion operators, string operators. This simple and modular structure allows the possibility of shifting functions from soft to firm or hard (see illustration 2).

The control structure of the machine appears on illustration 3.

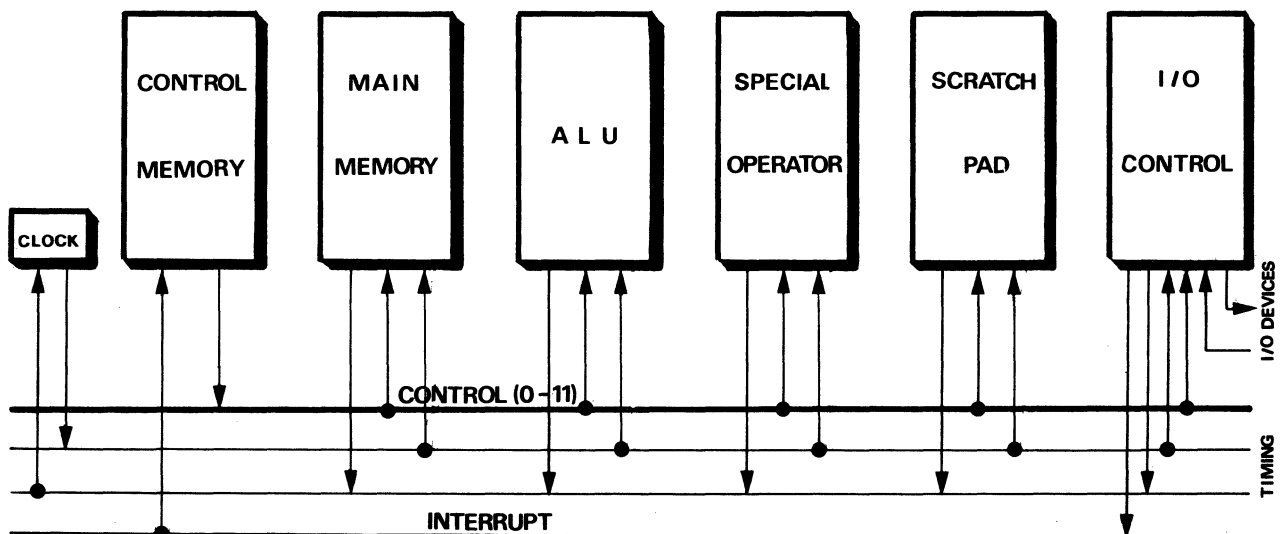
The control bus includes:

1. 12 control lines emanating from the data register of the control memory unit. They are gated to all the logical modules. They select a module and specify an operation code in 250 nsec. Whenever an operator module requires more than 250 nsec, another one may be accessed or initiated, resulting in parallel execution.
2. 9 internal timing lines: clock, test for condition, interrupt management.

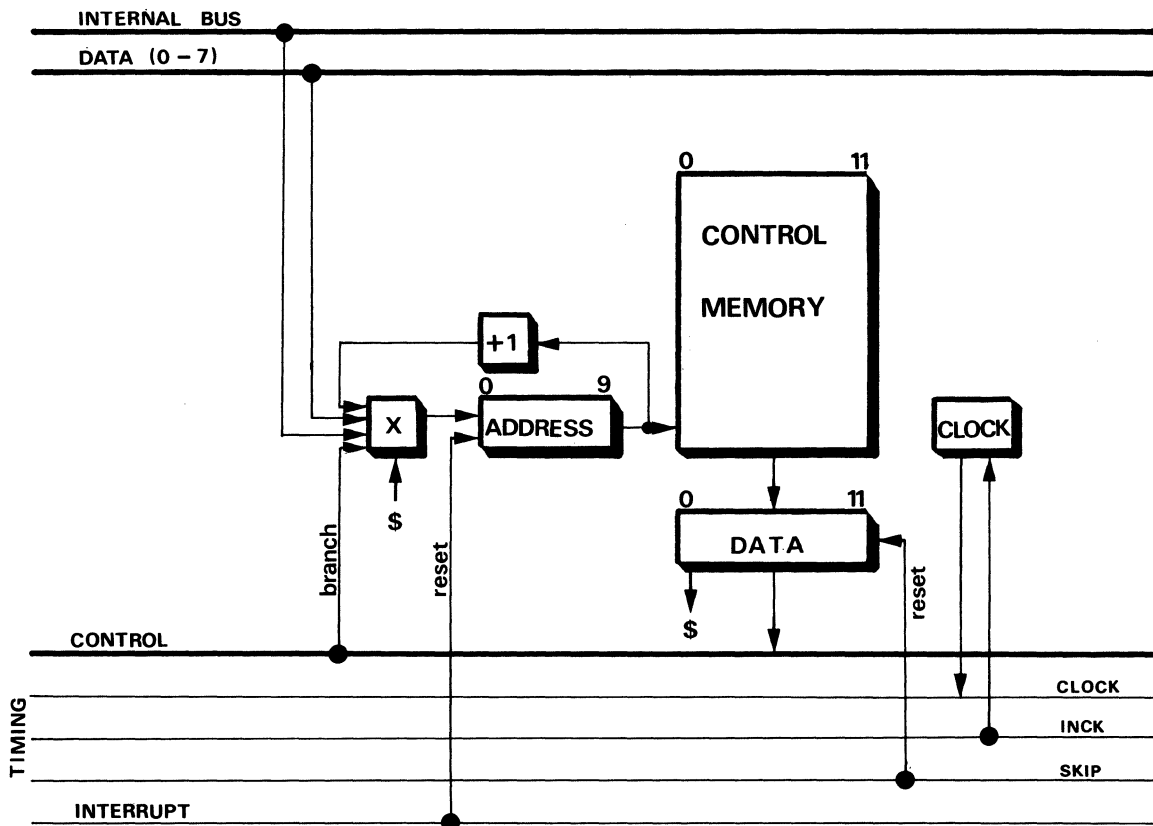
The control unit organization appears on illustration 4.

The control memory is addressable in four-word blocks (12 bit words). Its contents define the instruction set for the teleprocessing application considered. It includes a standard instruction set and specialized primitives, such as micro-programmed multiplexer channel-control, interrupt management, and communications control functions.

A real-time monitor performs task scheduling, and buffer and queuing management. It also handles user communications through a console, and provides inter-terminal transfers. The system's flexibility is used to tailor its architecture to the application. In particular, identical configurations can be interfaced successively to a number of host processors (multi-procedure facility). Intelligent or plain emulator versions allow the system to be plug-to-plug compatible with a large number of commercial hard-wired front-ends. It is also being used as concentrator, and for



**ILLUSTRATION 3 . THE CONTROL FLOW**



**ILLUSTRATION 4 . CONTROL UNIT ORGANIZATION**

the support of special peripherals. Capabilities under development are: general-purpose stand-alone facilities, multi-device file system, additional terminal support, user-microprogramming facilities, improved host-based programming aids.

This type of microprogrammed front-end processor may implement many operating system functions, and facilities, of the host processor, as its "intelligence" level increases. This may occur without any basic structural change, usually by expanding the software facilities. This achieves dual cost benefits:

1. An evolutionary front-end on a fixed hardware structure.
2. Reduced host processor's time consumption, as more of its functions get shifted to the front-end.

The increase in intelligence of the terminal is particularly

important: by smoothing the man-machine interface, it increases the human efficiency in using the front-end and the host processor systems.

#### PROSPECTS FOR A MICROPROGRAMMED FRONT-END ARCHITECTURE

The field of front-end processing has been expanding very rapidly. It has been shown how the increased complexity of a front-end system is handled in firmware, hardware, or software modules. Shrinking LSI costs will favor flexible micro-programmed systems, such as the one outlined here. Although many other criteria will eventually affect the marketability of such systems, such a modular and dynamically changeable architecture presents the best prospects for an efficient and flexible implementation.

# DESIGN OF A FULLY VARIABLE-LENGTH STRUCTURED MINICOMPUTER

Z. G. Vranesic  
V. C. Hamacher  
Y. Y. Leung

*Departments of Electrical Engineering and Computer Science  
University of Toronto  
Toronto, Canada*

## ABSTRACT

Binary-based and fixed-length structure computers are often inconvenient and wasteful of resources. In this paper we present a design for a fully variable-length structured minicomputer. Since all parameters (instructions and data) are unrestricted in length, their boundaries and interpretation are effected by special delimiter codes. For practical reasons (dictated by current technology) the machine utilizes a binary-coded decimal number representation.

## I. INTRODUCTION

Present day digital systems show a prevalence of binary, fixed-length structures. This is dictated by the technological ease of implementation, low cost and high reliability. Yet there are a large number of applications where the binary base and fixed-length organization are inconvenient and often wasteful of resources.

Decimal and variable-length data have been implemented in differing degrees from the IBM 1620 era [1] to one of the latest minicomputers, the CIP/2200 [2]. However, most of these machines have achieved these features in an "added-on" fashion in a structure that mainly offers conventional binary, fixed-length operations. The recently reported B1700 computer [3] reflects an attempt to get around the difficulties imposed by fixed-length constraints by providing a highly flexible, reconfigurable structure, where specific lengths may be defined as run time parameters.

In this paper we propose the design of a relatively small-scale decimal, variable-length machine whose structure evolves solely from those two features. In a sense, the work reported here can be interpreted as an elaboration or feasibility study on some conjectures made recently by Foster [4] concerning the architecture of the average computer of the year 2000. The design study described in more detail in the following sections in fact supports the feasibility of the basic concept even in terms of present day technology.

## II. MACHINE ORGANIZATION

In order to provide the variable-length characteristic for data, OP-codes and addresses, it is necessary to employ some "length delimiters". Thus it is apparent that a truly binary machine could not be constructed to meet such requirements, since the range of available digits (0,1) leaves no spare codes which could serve as delimiters. Hence we must turn to a higher base system, which for practical reasons might be binary coded.

Our choice is the decimal system with binary coded implementation. This provides us with six codes (other than 0-9) for use as delimiters. We will call them

$D = \{\alpha, \beta, \gamma, \delta, +, -\}$ .

The machine has a random-access memory with the capacity of 100,000 digits, addressable to the digit (0-99,999).

## NUMBER AND CHARACTER REPRESENTATION

In order to represent real numbers of the form  $N \times 10^e$ , both  $N$  and  $e$  are expressed as a sign followed by a 10's complement value. The exponent  $e$  is stored first, followed by the significant digits  $N$ , both number fields occurring low-order digits first.

For example  $+318.27 \times 10^{-12}$  is represented and stored in memory as  $-68+72813^*$ , which is equivalent to  $31827 \times 10^{-14}$ . The decimal point is always implied at the low order end of  $N$ . Note that  $*$  could be any delimiter.

Integer form is used for addressing purposes only ( $e = 0$  and it is not shown explicitly) and it is recognized as such from the context of instructions. We will refer to  $\{\pm\}$  followed by a string of digits as a number field or address, depending on the context, and use the name number or real number to refer to two successive number fields.

It is important to observe that the low-order digits are stored first, because the arithmetic unit must be at least partly serial, to enable it to handle arbitrarily long numbers.

The ASCII character set is represented directly using 2 4-bit digits per character. Any two digit delimiter not in the ASCII set, referenced in this paper as  $\square$ , is used as the character string delimiter. In general, the address of a character string, number, address, or instruction is the address of the leading delimiter, since this delimiter usually describes some property of the information to follow.

## INSTRUCTION SET

The machine has thirty-one instructions, including four rudimentary I/O instructions. Instructions are delimited by  $\alpha$ . An unsigned decimal opcode follows the leading  $\alpha$  and its end is indicated by any single digit delimiter. All instructions except HALT have an operand list which in some cases is preceded by a parameter  $K$ . The delimiter  $\delta$  indicates that  $K$  is present, and the possible values for  $K$  are integers equal to or greater than 0. The  $K$  value may be referenced by any of the addressing modes of Table 1. The interpretation of the delimiter set when used to separate items of the operand list is shown in Table 1. Table 2 gives the complete instruction set. In instructions where the parameter  $K$  is called for, it may be omitted if  $K = 1$ ; there being no ambiguity, since  $A$  can never be immediate data. The number of operands is variable in ADSB and MUDI instructions.

A few examples should clarify the appearance in memory of complete instructions, and give an idea of instruction execution.

(i)  $\alpha$ MVN $\delta$ 2+419 $\delta$ -97+32 $\alpha$ <sup>-21</sup>  
 This instruction inserts the real number  $23 \times 10^4$ , represented by 2 number fields, into the memory starting at the address 914; while  $\alpha$ MVN $\delta$ 4+419-0001 $\alpha$  moves the four number fields (2 real numbers or 4 addresses) starting at the address stored at location 1000 (indirect mode) into

914. If the number field (address) stored at 1000 has a "-" leading delimiter, then another level of indirection is indicated.

(ii)  $\alpha$ CPC+001 $\delta$ D<sub>1</sub>D<sub>2</sub>D<sub>3</sub>D<sub>4</sub>...D<sub>n-1</sub>D<sub>n</sub>  $\square \square \alpha$   
 $\frac{D_1 D_2}{C_1} \frac{D_3 D_4 \dots D_{n-1} D_n}{C_{n/2}}$

compares the character string  $C_1 C_2 \dots C_{n/2}$  with the one stored at 100 and sets the 2-bit condition vector in the CPU to 00 if they are equal, and to 11 if they are not; while  $\alpha$ CPC+001 $\gamma$ 456+27 $\alpha$

TABLE 1  
 Interpretation of delimiters in instructions

Delimiter	Addressing Mode or Function	Operand Form
+	direct	unsigned integer address.
-	indirect	unsigned integer address.
$\gamma$	indexed	unsigned integer address (the location of the index), delimited by + or - indicating a direct or indirect address to follow.
$\delta$	immediate	number, address or character string, appropriately delimited.
$\alpha$	instruction delimiter operation change (indicates SUBTRACT instead of ADD and DIVIDE instead of MULTIPLY in the ADSB and MUDI instructions.)	
$\beta$		

TABLE 2  
 Instruction Set

$\emptyset$ P CODE	$\emptyset$ PERAND LIST	DESCRIPTION
MVN	K,A,B	Move number fields (up to Kth delimiter) from B to A
MVC	K,A,B	Move character strings (up to Kth delimiter) from B to A
MVD	K,A,B	Move K digits from B to A
CPA	A,B	Compare addresses at A and B
CPN	A,B	Compare numbers at A and B
CPC	A,B	Compare character strings at A and B
CPD	K,A,B	Compare digit list at A and B
AND	K,A,B,C	Logical "AND" of K digits at B and C into A
$\emptyset$ R	K,A,B,C	Logical " $\emptyset$ R" of K digits at B and C into A
CMPL	K,A,B	Logical "complement of K digits at B into A
TRCT	K,A	Truncate number at A to K significant digits
CLR	K,A	Clear K digits starting at A
MVPN	K,A	Move pointer at A over Kth number field delimiter
MVPEN	K,A	Move pointer at A to the end of the Kth number field
MVPC	K,A	Move pointer at A over to Kth character string delimiter
MVPEC	K,A	Move pointer at A to the end of the Kth character string
ADSB	A,B,C,...	Add/Subtract B,C,... and put in A
MUDI	A,B,C,...	Mult./Div. B,C,..., and put in A
ADSB $\alpha$	A,B,C,...	Add/Subtract addresses (single number field)
CMPN	A,B	9's complement of the number field starting at B into A
BRZ	A	} conditional branches to A Br zero Br nonzero Br negative Br positive
BRNZ	A	
BRN	A	
BRP	A	
BR	A	
BR	A	Unconditional branch to A
JMPS	A	Subroutine linkage to A
HALT		stop
IN	K,A	I/O w.r.t. device K;
$\emptyset$ UT	K,A	transfer 16 bits of data
BIN	K,A	I/O w.r.t. device K;
B $\emptyset$ UT	K,A	transfer 8 bits of data

compares the character string at  $[[654]+72]$  where [...] indicates "the contents of".

- (iii)  $\alpha\text{MVPN}\delta 21+2\alpha$  takes [2] as an address and, assuming [2] points at a number field delimiter, increases the value of [2] until it points at the 12th number field delimiter from the starting point. This would allow [2] to now point at the 6th number down the list from the starting number. This provides the means for accessing variable length number or character strings in a list of such items where the programmer knows explicitly only the address of the first item.

In the above examples, we have used appropriate mnemonics for the OP-codes, but they are actually specified in memory by unsigned integer codes.

### III. HARDWARE DESIGN

Since all parameters may be variable in length, a fully parallel design of the machine cannot be achieved. It is apparent that serial by digit structure would be the simplest solution in terms of hardware costs. However, in order to attain a reasonable processing speed, some degree of parallelism must be

introduced.

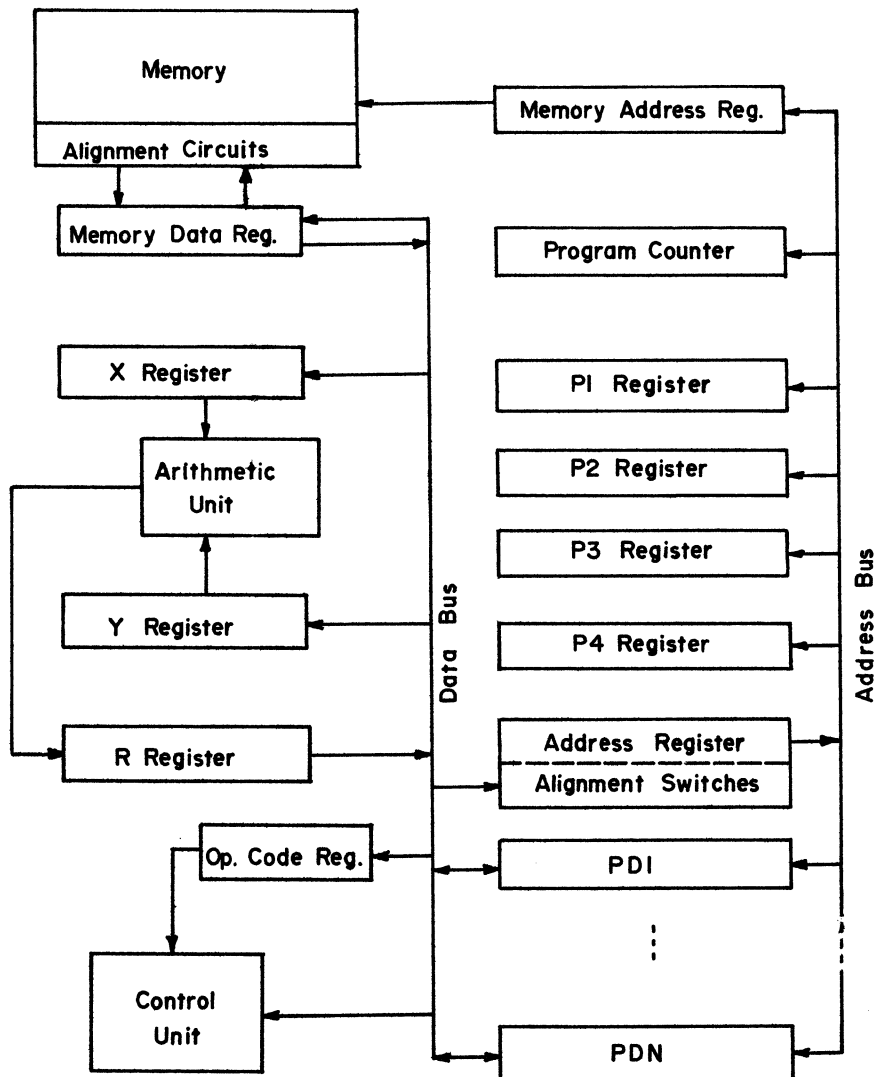
In our prototype design we have chosen serial processing of four-digit (16 bits) blocks of data. Figure 1 shows the block diagram of the machine.

Memory has a 16 bit word length and its addressing is arranged in a  $4 \times 25 \times 1000$  digit pattern, giving a total capacity of 100,000 digits. It is digit-addressable, necessitating two internal read cycles if the address is not 0 or divisible by 4. In order to avoid alignment difficulties on the data bus and in the 4-digit parallel arithmetic unit, the memory includes alignment circuits so that the memory data register always contains the addressed digit plus the three digits that follow. Thus it is not necessary to impose any boundary alignment conditions on the programmer for storage of data in the memory.

Internal sequencing and serial control of instruction execution when the operand length exceeds 4 digits is regulated by the pointer registers P1, P2, P3 and P4, each being a 5-digit counter-register.

All addressing is carried out via a 5-digit address bus. Since addresses are obtained directly from instructions they are not necessarily correctly aligned on the data bus. This is remedied by assembling all addresses in the address register which

FIGURE 1  
Block Diagram



includes the required alignment switches.

Peripheral devices PD1,..., PDN are addressed through the low order digits of the address bus, with data transfer handled by the data bus.

#### IV. PROGRAMMING CONSIDERATIONS

Consistent with the theme of Foster's [4] brief sketch of the average computer of the year 2000 which "... will-be a monoprocessor doing its own I/O, - most probably be privately owned and monoprogrammed, - be an interpretive engine capable of executing directly one or more high-level languages...", it is claimed that although we do not interpretively execute several high-level languages, the instruction set of Table 2 makes possible efficient processing on a "one-shot" basis of relatively small user programs. This is a reasonable goal for a small, general, privately owned and monoprogrammed computer in any event. The efficient processing we mentioned above is from the programmer standpoint. This means that the machine language, represented in some assembly form, should have instructions and formats that make coding of normal problems somehow natural and concise.

#### MATRIX MULTIPLY ROUTINE

We first present a complete program to multiply two matrices of real numbers. All matrix entries are of variable length, so normal indexing would not work on any machine, and the equivalent program in a fixed word length structure would be somewhat clumsy and unnatural.

The program performs the computation  
 $C = A \times B$  where A is ID rows by JD columns,  
 B is JD rows by KD columns,  
 and C is ID rows by KD columns.

Matrices are stored in column order and the program variables for the matrix dimensions are the same as above.

Assuming that the matrices A and B have been loaded in core and ID, JD, and KD have been appropriately initialized, the program is:

```

ADSBA II←ID+ID      ;increment step for PT1
ADSBA K←-KD        to access successive
MVN   JV←0         row entries.
MVN   PT3←#C       ;load address of C into
KL00P: MVN   M←0    PT3.
ADSBA I←-ID
IL00P: MVN   PT2←#B
MVPN  JV,PT2      ;sets PT2 to appropriate
MVN   PT1←#A      column of B.
MVPN  M,PT1       ;sets PT1 to appropriate
ADSBA J←-JD       row of A.
MVN   2,'PT3←0.0  ;clear ci,k(initial length
                    unimportant.)
JL00P: MUDI TEMP←'PT1*'PT2 ;ai,jx bj,k
ADSBA 'PT3←'PT3+TEMP ;accumulate into ci,k
MVPN  2,PT2       ;move PT2 across 2 delimi-
                    ters to b(j+1),k
                    ;move PT1 to ai,(j+1)
MVPN  II,PT1
ADSBA J←J+1
BRN   JL00P
MVPEN 2,PT3       ;move PT3 to next C entry
ADSBA M←M+2
ADSBA I←I+1
BRN   IL00P
ADSBA JV←JV+JD+JD ;sets B column accessing
ADSBA K←K+1       variable.
BRN   KL00P
HALT

```

In this program, and in the remainder of this section, we have used a suitable assembler notation for

the parameter and operand lists for instructions. For example, #C refers to the address of C, and 'PT3 indicates indirect addressing through location PT3. There are no macro references; and there is a strict one-to-one correspondence between the lines in the program and machine instructions.

The previous example was concerned with arithmetic operations and array accessing. We now illustrate some aspects of non-numerical programming. The example chosen can be taken as a model of some aspects of symbol table manipulation in a language processor. The main idea here is to illustrate the ease of building and searching tables of variable length mixed data types.

#### SYMBOL TABLE MANIPULATION

A particular type of character string made up of ASCII symbols  $\underbrace{A,B,\dots,Z}_{<A>}, \underbrace{;,:,\$}_{<D>}$  is to be processed.

A syntactically correct string must start with a member of <A> and end with a member of <D> followed by \$, with no other occurrences of \$, and with all other occurrences of members of <D> isolated by members of <A>.

There are also some semantic rules that must be met. First, we need some definitions. Each occurrence of a member of <D> will be said to "terminate" the previous contiguous substring of members of <A>, and the class name <LABEL> will be used to describe any such contiguous substring of members of <A>. Now, a syntactically correct string is also semantically correct if all <LABEL>'s terminated by ":" are unique and any <LABEL> terminated by ";" also appears in the string terminated by ":".

Examples of correct and incorrect strings are:

- (i) START:L00P:CTR:L00P;OUT:\$ is both syntactically and semantically correct.
- (ii) A:A;SRCH;COMP:SRCH:A:\$ is both syntactically and semantically incorrect (see the underlined places).

The processing to be performed on these strings is as follows: Build a table in core of all unique <LABEL>'s with an address associated with each. If the <LABEL> first occurs terminated by ":", the address is provided from the contents of a word addressed as L0CCTR; otherwise, the 5-digit word 00000 is associated with the <LABEL>. This "dummy" address will be replaced by the correct value from L0CCTR when the <LABEL> later occurs terminated by ":".

There are two subroutines used in the program which we will list in detail. One, called TBLSCH, is used to search the table for the occurrence of the <LABEL> currently in 'BUFF. The locations T0P1 and B0T1 are pointers to the top and bottom of the table, respectively; and PT1 is a pointer location for accessing the table entries. On exit, put "Y" in ANS if the <LABEL> is found, and leave PT1 pointing at the lead delimiter for the matching <LABEL>; otherwise, put "N" in ANS. The routine is accessed from a JMPS instruction which puts the return address in the first location, TBLSCH, in the routine.

The coding is:

```

TBLSCH: DA   5           ;assembler command to
MVN   PT1←B0T1         establish a 5-digit "return
CPA   T0P1←B0T1       field."
BNZ   CHECK           ;go to CHECK if table non-
MVD   2,ANS←"N"       empty.
BR    'TBLSCH
CHECK: CPC 'BUFF←'PT1  ;compare LABEL in 'BUFF
BZ    F0UND           with one in table.
MVP   PT1             ;move PT1 to start of next
ADSBA PT1←PT1+2       <LABEL> in table
MVPN  PT1
ADSBA PT1←PT1+1

```



```

CPA PT1←TØP1 ;has whole table been
BNZ CHECK searched?
MVD 2,ANS←"N"
BR 'TBLSCH
FOUND: MVD 2,ANS←"Y"
BR 'TBLSCH

```

The second routine, called TBLINS, inserts the <LABEL> in 'BUFF onto the top of the symbol table and associates the address in PARAM with it. The pointer TØP1 is adjusted appropriately.

```

TBLINS: DA 5
MVN PT1←TØP1
MVC 'TØP1←'BUFF ;add <LABEL>
MVPEC PT1
MVD 2,'PT1←'□□'; ;insert character delimi-
ADSBA PT1←PT1+2 ter
MVN 'PT1←PARAM ;insert associated address
MVPEN PT1
MVD 'PT1←+' ;insert number field deli-
ADSBA PT1←PT1+1 miter
MVD 2,'PT1←'□□" ;insert table-top delimi-
ter
MVN TØP1←PT1 ;adjust table-top pointer
BR 'TBLINS

```

Although we have only presented two of the sub-routines used in the complete program, the type of coding used at the assembler level for non-numeric processing should be evident. The complete program required 110 instructions, including the subroutine coding.

Due to the radically different structure, it is difficult to compare our machine with standard minicomputers. Meaningful comparisons will become possible only as a result of extensive experience with it. The machine was simulated and some interesting observations made. For example, the above matrix multiply routine was found to require 400 digits of storage with the delimiter density of 25%.

#### V. CONCLUSIONS

We have described the design of a fully variable-length general purpose computer. In order to assess the feasibility of such machines it is essential to take a close look at advantages gained and difficulties that might be encountered.

Based on a number of programs that we have written, it is apparent that programming presents fewer difficulties than one usually encounters with standard minicomputers.

Limits on computational accuracy, size of operand labels and data as well as the alignment requirements, are non-existent from the programmer's point of view by the very nature of the machine.

In order to determine the physical feasibility of such machines, we have completed the design on the basic circuit level (using standard TTL MSI components). As a result we have found that the hardware complexity and cost place the machine in the price range of typical minicomputers. Simulator runs have been used to verify the logical correctness and adequacy of the selected instruction set, as well as to obtain an evaluation of memory and cycle time requirements.

#### VI. ACKNOWLEDGEMENT

This research was partly supported by the National Research Council of Canada.

#### VII. REFERENCES

- 1 IBM Reference Manual, 1620 Data Processing System, IBM, 1960.
- 2 CIP/2200 Reference Manual, Cincinnati Milacron Company, Process Controls Division, Lebanon, Ohio, April 1972.
- 3 W.T. Wilner, "Burroughs B1700 memory utilization," Proceedings of FJCC, 1972, pp. 579-586.
- 4 Caxton C. Foster, "The Next Three Generations," Computer, Vol. 5, No. 2, March/April 1972.



# HAPPE

## HONEYWELL ASSOCIATIVE PARALLEL PROCESSING ENSEMBLE

Dr. Orin E. Marvel  
*Honeywell Inc.*  
*Aerospace Division*

### ABSTRACT

Many problems, inherent in air traffic control, weather analysis and prediction, nuclear reaction, missile tracking, and hydrodynamics have common processing characteristics that can most efficiently be solved using parallel "non-conventional" techniques. Because of high sensor data rates, these parallel problem solving techniques cannot be economically applied using the standard sequential computer.

The application of special processing techniques such as parallel/associative processing are still resisted because it is a change from the norm. Past implementations utilized special hardware, custom circuits and complex designs. The Honeywell Associative Parallel Processing Ensemble (HAPPE) was built to demonstrate the basic simplicity of hardware concepts inherent in parallel associative processing. HAPPE is implemented with the same standard circuit building blocks (MSI's, ROM's, and RAM's) that are used in all conventional computers. By using standard building blocks a long time objection to associative memory systems, that of requiring special purpose (low usage) circuits is overcome.

The parallel/associative processing element can be both powerful and versatile. The HAPPE architecture proves that one processor element can perform both correlation (associative) and arithmetic processing. The HAPPE demonstrator has become a valuable tool in training system designers and programmers to recognize that new "thinking" can be applied to advanced processing system implementations.

### Introduction

Problems associated with systems such as radar tracking and discrimination, air traffic control, weather prediction, nuclear reactor control and hydrodynamic prediction have common characteristics. Each of these problems exhibits a high degree of parallelism; that is, many sets of data must be evaluated, manipulated and reduced by the same computing process as rapidly as possible and preferably simultaneously.

Those systems requiring a real time problem solution should take advantage of this parallelism by

solving each problem set in a simultaneous manner. The architectural solutions to these problems have led to considerable discussions on special processing techniques particularly on pipeline processing versus parallel associative processing. Numerous papers have been written on the subject. It has been shown that with the technology available today (2), many of the problems discussed earlier can be solved by the pipeline processor in a more economical way. These problems, however, require that the input sensor data always occur in the same order. As the input data becomes more unordered or random, the parallel associative processor becomes the only candidate available for real time processing (1), (4).

The Honeywell Associative Parallel Processing Ensemble, HAPPE, was built with standard building blocks to demonstrate the hardware concepts and inherent simplicity and computing power of parallel associative processing and the processing element. Its functional organization is designed to accommodate both I/O associative processing and simulated track processing.

During track processing, the input space built into the hardware is divided into two random sets. Thus if an input sensor or target has provided an input in one of the two sets, for associative processing and then switches to the other set, the track processing associated with the first set of data will be delayed a cycle. However, we still have no prior knowledge of when the data arrives at the processing system. The HAPPE demonstrator proves that one processor element (hardware entity) can perform both correlation (association) and arithmetic processing.

The HAPPE processor is implemented with the same standard building blocks (MSI's, ROM's and RAM's) that are used in more conventional computers. This overcomes a long-time objection to associative memory systems that require special purpose building blocks, such as complex one-of-a-kind large scale integrated circuits.

### Background

The evolution of the parallel associative processor is shown functionally in Figure 1. Reference (3) describes the historical evolution of associative processors.

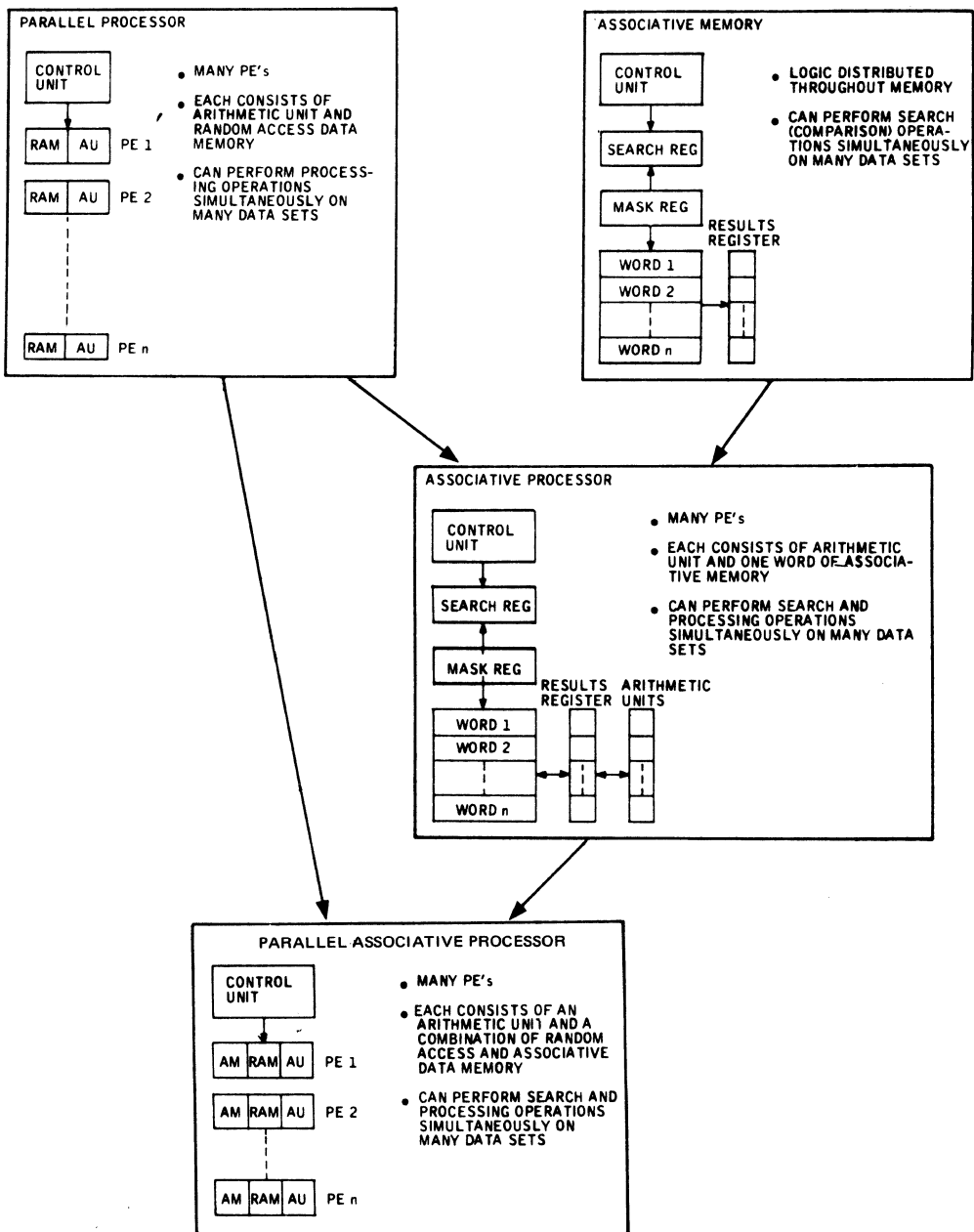


Figure 1. Parallel Processor/Associative Processor Relationships

The parallel processor is characterized by a control unit and a number of processing elements that perform simultaneous operations. The control unit to processing element bus is used primarily in a sequential manner to either carry inputs, commands or outputs.

The Associative Memory is primarily a storage facility with logic added to each memory position to perform associative searches. These searches are classified primarily as input searches and output searches.

When performing an input associative operation, an input data set is processed by means of the following search operations:

What stored data is less than the input?

What stored data is greater than the input?

What stored data is within a delta limit of the input?

No stored data is within a delta limit of the input.

The input operation is primarily a matching operation between data sets stored in the processors and randomly occurring input data sets.

The output associative operation searches the associative memory to determine:

The stored data with the minimum value.

The stored data with the maximum value.

All stored data within a given sector of space.

All stored data outside a given sector of space.

The output operation is used to provide the user with the exact information he needs to make an operational or control decision.

The associative processor is really closer in functional execution to the associative memory than it is to the parallel processor. Primarily then, it is an associative memory with minimal processing capability associated with each memory word. The STARRAN is an example of the associative processor.

As we move up the functional ladder from associative memory, to associative processor, to parallel associative processor, we are asking the system to perform more and more processing for every associative or correlation operation. Thus the parallel associative processor is aimed at the application requiring a high random sensor input rate, accompanied by a large number of sensor dependent computations performed on each input, in real time. The HAPPE system was designed to demonstrate the functional sophistication afforded by parallel associative processors.

### HAPPE System

The HAPPE system demonstrates the use of associative parallel processing in solving the complex problem of target tracking using a modern phased

array radar as a sensor. Phased array radars provide a modern tracking system with the ability to electronically point and shape one beam, or a multiplicity of beams, and to steer them at electronic speeds. Target location coordinates in phased array radar systems are specified by range and beam number, rather than by range, azimuth, and elevation. The beam number incorporates both azimuth and elevation of the target. This data gathered at electronic speeds, demands close coupling with its processing operation in order to meet the workload and real time restrictions of this high performance system.

The HAPPE organizational design consisting of two global control sections and a number of redundant processing elements is primarily based on the following observations:

- (A) Radar data is constantly coming into the system without interruption. First, radar data comes from random beams in the first half of the sensor space and then from the second half of the sensor space.
- (B) Target data is associatively assigned to processing elements then an arithmetic program (the track processing) is completed using the correlated data before the next correlation can take place.
- (C) A standard arithmetic building block (the "181") performs the arithmetic add, subtract, and logical operations; as well as the less than, greater than, and equal to correlation operations.
- (D) The radar tracking algorithms can be performed without built-in multiply or divide capability.

An analysis of the requirements stated above led to the architecture shown in Figure 2. This block diagram resembles the parallel associative processor of Figure 1. But, operationally HAPPE uses a different philosophy from any other associative processor. In the HAPPE system, the processing element controls its own mode and process state. The control units do nothing but repeatedly broadcast their algorithms or commands and data, never knowing whether any elements are reacting to these commands or not.

In comparing HAPPE and PEPE, the HAPPE system contains two control units and one processing element unit, while the PEPE system contains three control units and three processing element units. In HAPPE, the control units alternately control the element unit; but in PEPE, each of the control units controls only its associated element unit.

The processing elements operate in two modes-- the correlation mode and the arithmetic mode-- and within each mode two states-- active and inactive. Of course the system must contain a master reset which sets every element to the correlation mode and active state.

When an element is in the correlation mode and active state, it performs all commands and accepts all data from the correlation control unit.

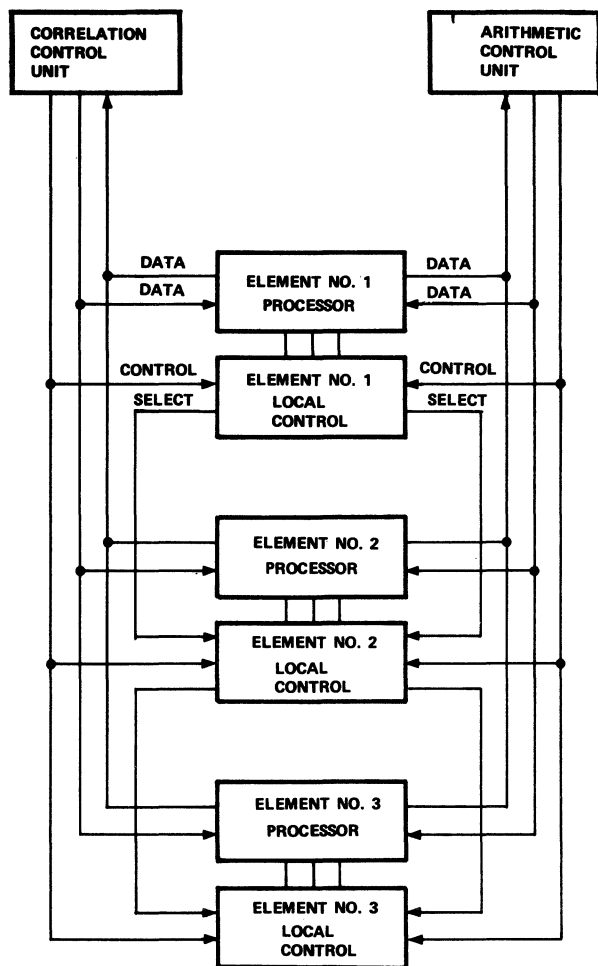


Figure 2. System Block Diagram

Any element in the correlation mode and inactive state can perform the following commands from the correlation control unit:

- Activate All
- Master Reset
- Select Next Inactive.

When an element is in the arithmetic mode and active state, it performs all commands and accepts all data from the Arithmetic Control Unit. Any element in the arithmetic mode and inactive state can perform the following commands from the Arithmetic Control Unit:

- Activate All
- Master Reset.

The correlation control unit provides the following commands:

- Deactivate on Not Equal to
- Deactivate on Less Than
- Deactivate on Greater Than
- Load Local Memory to A Register
- Load Local Memory to B Register

- Store Global Data to Local Memory
- Activate All
- Switch Modes
- Master Reset
- Select Next Active
- Select Next Inactive.

The Arithmetic Control Unit provides the following commands:

- Load Local Memory to A Register
- Load Local Memory to B Register
- Store A Register to Local Memory
- Add
- Subtract
- Switch Modes
- Deactivate on Not Equal to
- Shift Left
- Output
- Master Reset
- Activate All.

A block diagram of the processing element is shown in Figure 3. The radar processing problem requires that a correlation process assign targets to processing elements and then process the data associated with each target. The processing element resembles any minicomputer containing two registers, an arithmetic unit, and a scratchpad memory. The element differs by containing a local activity and mode control which can be modified by the comparison outputs of the arithmetic unit.

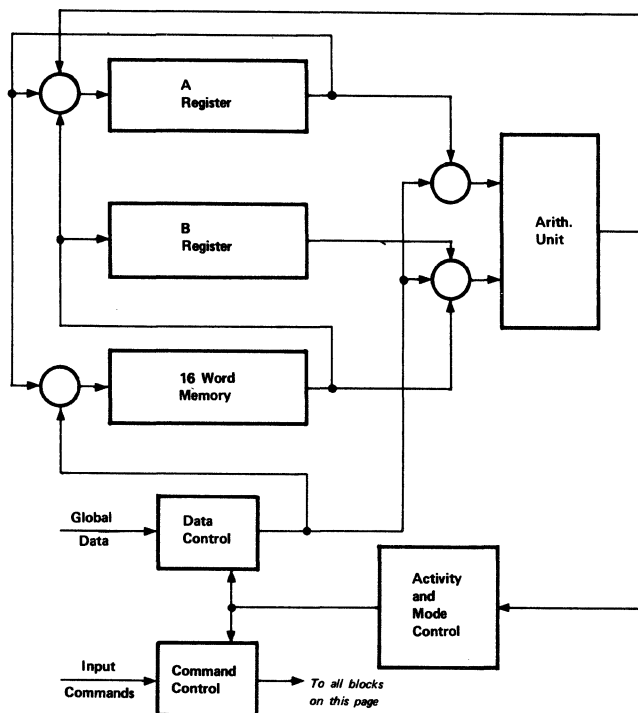


Figure 3. Processing Element Block Diagram

Each processing element operates like a correlation unit when in the correlation mode (accepting commands from the correlation control unit) and like an arithmetic unit when in the arithmetic mode (accepting commands from the arithmetic control unit). Each HAPPE processing element performs add, subtract, and shift operations, as well as equal to, less than, and greater than operations. During the arithmetic mode, the track update programs are performed and during the correlation mode, the target data assignment programs are performed.

Demonstrator Description

The HAPPE demonstrator under discussion and illustrated in Figure 4, consists of three 4-bit processing elements. The demonstrator operates with two beam numbers (a realistic system would have about 250) and two incoming targets that have an arbitrary decreasing range assignment of 15 to 0 units. A realistic system would look at up to 1000 targets coming in from about 150 KM.

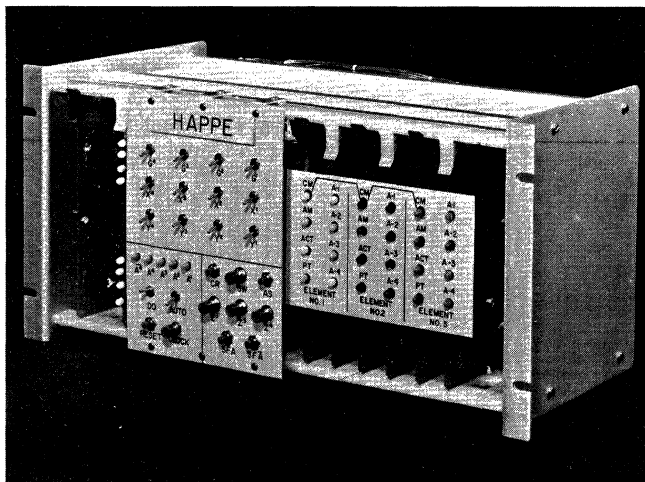


Figure 4. HAPPE Demonstrator

The problem solved by the demonstrator consists of recognizing the beam number of the target, correlating the range of the target with a range gate in each processing element, and then performing an arithmetic subroutine on the correlated data. A pictorial of the simulated system is shown in Figure 5 and an example of the radar return for an incoming target is shown in Figure 6. An example of a range gate computation for one target can be seen in Figure 6, with the R (minimum) set at 10 and the R (maximum) set at 13. A target moving through this range gate will cause four between-limit matches.

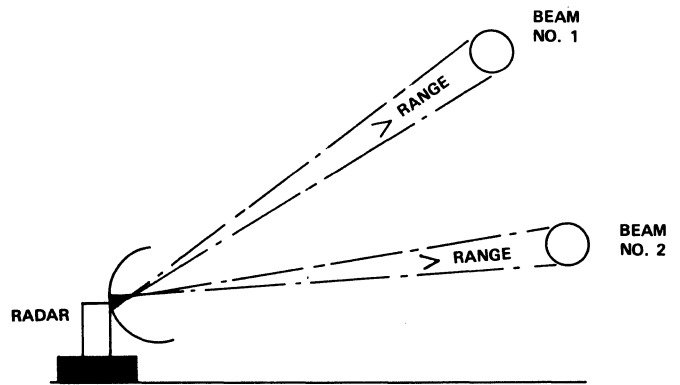


Figure 5. Physical Model

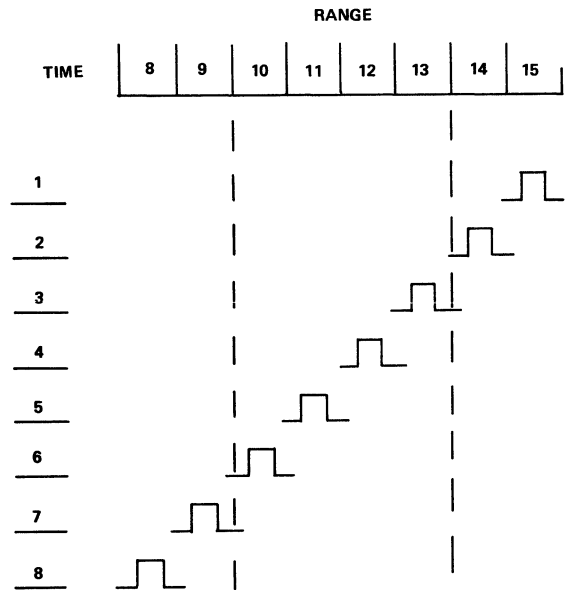


Figure 6. Target Movement

Figure 7 describes the control panel, the initialization of the demonstrator, and the results of the target tracking program. For example, processing element 1:

- A. During the set up of the demonstrator, the beam number (beam 1) is stored in scratchpad location (X1, Y1), the lower gate (Range 2) in (X1, Y2), and the upper gate (Range 8) in (X1, Y3).
- B. During the operation of the simulation, the processing element only reacts to the target, while it is in the range of 8 to 2.
- C. At the end of the operation, the element has seen the target seven times.

## CONTROL PANEL (Switches Up are On)

Global Data Lines ( $G^1, G^2, G^3, G^4$ ) - Provides binary coded data to all elements which are active and in the correlation mode.

X Data Lines ( $X^1, X^2, X^3, X^4$ ) - Selects the X data location of the element memories.

Y Data Lines ( $Y^1, Y^2, Y^3, Y^4$ ) - Selects the Y data location of the element memories.

D. O. - Allows the wired "OR" outputs of the element "A" registers (which are active) to be displayed on the panel lights ( $A^1, A^2, A^3, A^4, \bar{A}^4$ ).

Auto - Sets the control units in the auto clock or single step program mode.

Reset - Resets the counters of the control units to restart the program.

Clock - Allows the control units to be single stepped through their programs when the auto switch is off.

CR - Sets all elements to the correlation mode.

SW - Switches the mode (correlation or arithmetic) of all the elements that are active.

ACT - Sets all elements to active.

$Z_1$  - Clocks the data selected by global lines into the memory location selected by the X, Y switches (for all active elements in the correlation mode).

$Z_3$  - Clocks the data from the memory location selected by the X, Y switches into the A register (for all active elements in the correlation mode).

$Z_4$  - Clocks the activity flipflop when using the select commands

SFA - Activates the pointer for selecting the first active element in the correlation mode.

$\bar{SFA}$  - Activates the pointer for selecting the first inactive element in the correlation mode.

## SET UP OF DEMONSTRATOR

PUSH CR BUTTON

PUSH AS BUTTON

WRITE ( $Z_1$ ) 0 IN  $X_3, Y_1$

WRITE ( $Z_1$ ) 0 IN  $X_3, Y_2$

WRITE ( $Z_1$ ) 2 ( $G_2$ ) IN  $X_2, Y_1$

WRITE ( $Z_1$ ) 5 ( $G_3, G_1$ ) IN  $X_2, Y_2$

WRITE ( $Z_1$ ) 4 ( $G_3$ ) IN  $X_2, Y_3$

WRITE ( $Z_1$ ) 0 IN  $X_2, Y_4$

PUSH SFA AND  $Z_4$

WRITE ( $Z_1$ ) 1 ( $G_1$ ) IN  $X_1, Y_1$

WRITE ( $Z_1$ ) 2 ( $G_2$ ) IN  $X_1, Y_2$

WRITE ( $Z_1$ ) 8 ( $G_4$ ) IN  $X_1, Y_3$

PUSH SW BUTTON

PUSH SFA AND  $Z_4$

WRITE ( $Z_1$ ) 2 ( $G_2$ ) IN  $X_1, Y_1$

WRITE ( $Z_1$ ) 6 ( $G_3, G_2$ ) IN  $X_1, Y_2$

WRITE ( $Z_1$ ) 13 ( $G_4, G_3, G_1$ ) IN  $X_1, Y_3$

PUSH  $\bar{SFA}$  AND  $Z_4$

WRITE ( $Z_1$ ) 1 ( $G_1$ ) IN  $X_1, Y_1$

WRITE ( $Z_1$ ) 4 ( $G_3$ ) IN  $X_1, Y_2$

WRITE ( $Z_1$ ) 6 ( $G_3, G_2$ ) IN  $X_1, Y_3$

PUSH AS BUTTON

PUSH CR BUTTON

SELECT AUTO MODE

## RESULTS

### A. TARGET TRACKING PROGRAM

Element Number 1      7 in A-Reg.

Element Number 2      8 in A-Reg.

Element Number 3      3 in A-Reg.

### B. ARITHMETIC PROGRAM

(All Elements ( $X_2, Y_4$ ) three)

Push As

Select  $X_2, Y_4$

Push  $Z_4$

Read A-Register

### C. TO RESTART PROGRAM

Push CR Button

Push AS Button

Select  $X_3, Y_1$

Push  $Z_1$

Push  $Z_3$

Clear  $X_3, Y_1$

Push Reset



## Programs

The sample programs executed by HAPPE are typical of real-life problems and are described in the following:

- (A) In the correlation program, one of two beam numbers is compared with the beam number data stored in the elements. A between limits search on the range data provided by the simulator is then made. During the limit search, each of the three processing elements will have made use of its own range gate. At the conclusion of the correlation processing mode, those elements that have just performed correlations are switched to the arithmetic mode and those that have completed the arithmetic processing are switched to the correlation mode.
- (B) The arithmetic program updates and accumulates the number of between-limit hits in each of the elements. This program also demonstrates typical arithmetic operations which are required to calculate a new range gate.
- (C) The select highest and output program selects the one element which has the highest number of range gate hits and outputs this number to the control unit.

The page limitations of this paper have prevented the inclusion of the program flow diagrams. These can be obtained by contacting the author.

## Conclusions

The HAPPE demonstrator shown in Figure 4 was built to demonstrate several important concepts of parallel/associative processing through use of a simulated phased array radar processing system. These are:

- (A) That one processing element implemented with standard conventional logic circuits can perform both correlation and arithmetic processing.
- (B) That two control units operate simultaneously on different processing elements.
- (C) That the operation, initialization, mode switching, and activity of parallel processing elements can be easily controlled.
- (D) That the signal distribution and busing required to effectively operate a parallel associative processor uses standard circuits.
- (E) That the correlation and arithmetic programming takes advantage of standard programming methods with the addition of a small number of parallel control instructions.
- (F) That HAPPE has a built in "Fail Graceful" characteristic where one element can fail or be disengaged from the system and the rest of the system operate at a reduced target load.

- (G) That a low cost, three element, 4-bit data associative parallel processor can be built to demonstrate the operating characteristics of a large system. This demonstrator is also an excellent training tool for advanced architectures and the software required to make them operate.

## Bibliography

- (1) Hobbs, L. C., "Parallel Processor Systems, Technologies and Applications", Sparton Books, 1970.
- (2) Lloyd, G. and Merivin, W., "Analysis of Three Large Computer Systems", AFIPS National Computer Conference, June 1973.
- (3) Parhami, B., "Associative Memories and Processors: An Overview and Selected Bibliography", Proceedings of the IEEE, Vol 61, No. 6, June 1973.
- (4) Thurber, K. J. and Berg, R. O., "Applications of Associative Processors", Computer Design, Vol 10, pages 103-110, November 1971.



# A COMPUTER ARCHITECTURE AND ITS PROGRAMMING LANGUAGE

Mario R. Schaffner  
*Massachusetts Institute of Technology  
Cambridge, Massachusetts*

## 1. INTRODUCTION

Computer architectures and programming languages are traditionally developed independently. Through suitable computer architecture, for instance, one can attempt to speed up the processing of a stream of data and instructions, leaving to the software the burden of preparing these streams. Through a suitable programming language, one aims at efficiently describing many classes of problems, in a phrase-structure form that is machine independent. This approach has led to the ever-increasing application of computers, but it has also brought about a growing complexity in the software systems. As a consequence of the latter, there is a new trend toward extending hardware implementation to replace the software ones, especially in view of the new technology of large scale integration.

In the past, several computer architectures were suggested toward the attainment of a larger computing power, such as the Solomon computer [1], the Holland machine [2], a spatially oriented computer [3], and a fixed plus variable structure computer [4]. Then, two basic techniques appeared of more general applicability, and led to actual implementations: parallel processing, and pipeline execution [5]. In parallel processing, an array of similar processors work simultaneously on different data, under the control of the same control unit. The modularity of such an organization is attractive in many respects. However, the performance is heavily dependent on parallelism in the problems [6], and programming techniques need to be developed, for exploiting the potential capabilities of the computer and the inherent parallelism in the computations [5]. Whereas for particular problems parallel computers can achieve a throughput which is orders of magnitude larger than that of conventional computers, for general problems they face a performance degradation that increases with the number of processors.

Pipelining consists of the concurrent execution of the various stages of the processing by independent units connected in cascade. This concurrency can be implemented at different levels [7]; the overlapping of the processor and memory operations [8], and that of the steps of arithmetic operations [9] are examples. The theoretical limits of pipelining have been analyzed [10]; in practice, advantages depend upon the presence of a stream of similar tasks [11]. All modern large computers have some degree of parallelism and pipelining. In order to analyze their organization, instruction and data streams can be defined, and the management of requests and services considered [12]. In this context, computer architectures are, at first, classified as single-instruction single-data streams, single-instruction multiple-data streams, multiple-instruction single-

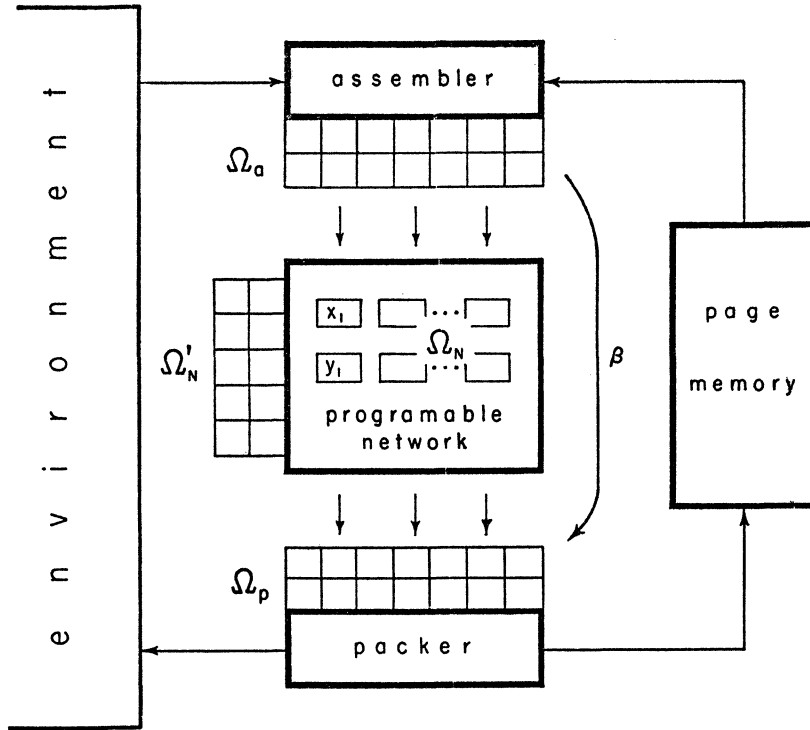
data streams, and multiple-instruction multiple-data streams.

The use of these computer architectures depends heavily on complex compilers, or interpreters. Compilation requires a preliminary run, generally produces no optimum codes and makes the debugging more difficult. Interpreters require a large memory space and produce a slow execution. For these reasons the question rises recurrently whether computers constructed to directly execute programs written in the user programming language could lead to a more efficient overall system [13]. The above question has prompted several works oriented to the hardware (or a mixture of hardware and software) implementation of the production of phrase-structure programming languages that are subsets of existing programming languages, or a slight variation of them. Some examples are: Anderson's [14] implementation of Algol 60, Bashkow's et al. [15] design of a Fortran machine, Weber's [16] implementation of EULER, Thurber's et al. [17] design of a cellular APL computer, the SYMBOL language and computer [18], and the APL implementation by Hassitt et al. [19]. All these studies show particular advantages in more closely relating the structure of the programming language and the structure of the computer hardware. However, no significant impact was made on the mainstream of computers, in which languages and hardware are developed independently. One can argue that in the above cases the languages used (at least basically) already existed and were developed independently of any particular architecture.

This paper shows a case in which computer architecture and programming language are not developed independently, but are treated as two isomorphic forms of representation of the same structure -- the abstract mechanisation of the processes as it is conceived by the user. The user models the desired process in the form of an abstract Finite State Machine (FSM), at a proper level, in terms of the elements of a language for describing FSMs. The description of this FSM constitutes a description of the desired process, but at the same time is also the specialized architecture of a hypothetical machine that executes that process. If a physical substratum (isomorphic with the language of the FSMs) is available, these hypothetical machines can be implemented, and the description of an FSM constitutes a program for this substratum. Such a substratum can be seen as an organizable computer. In this case, the distinction between hardware and software blurs.

This substratum, i.e. a programmable architecture, is outlined in section 2; the isomorphic programming language is described in section 3; and in section 4, results and implications are discussed.

# FIG 1



## 2. THE PROGRAMABLE ARCHITECTURE

The essential parts of this architecture are, Figure 1:

(1) a programmable network PN comprising an array  $\Omega_N$  of registers for holding a page of data, a second page array  $\Omega'_N$ , and programmable operational elements which can be connected to these registers with the related control circuitry for the execution of operations on the variables;

(2) a memory for holding pages of data, the structure of which can be programmed in accordance with the data structures of the processes;

(3) an assembler which, receiving a page from the memory and new data from the environment, assembles the variables of a process and program words (that describe networks performing the operations of the present state of the process) into a page register-array  $\Omega_a$ ; and

(4) a packer which, receiving a page from PN into a page register-array  $\Omega_p$ , provides the routing of output data to the environment, and the packing of the data needed in the future into the form of a page for the memory.

A page here is a self-sufficient set of data related to a job; in the memory, it contains the present variables of the process and a key word indicating the present state of the process; in the assembler and in PN, it also includes the new input data and the program words of the present state.

The basic page transfers can be described with the use of register transfer notations

[20] by the expressions

$$t_1 \alpha_1 \Omega_a + t_2 \gamma F_1(\Omega_N, \Omega'_N) + t_3 \delta_1 \Omega'_N \rightarrow \Omega_N$$

$$t_2 \gamma F_2(\Omega_N, \Omega'_N) + t_4 \delta_2 \Omega_N \rightarrow \Omega'_N \quad (1)$$

$$t_5 \alpha_2 \Omega_N + t_6 \beta \Omega_a \rightarrow \Omega_p$$

where  $F_1$  and  $F_2$  are functions executed by the programmable network;  $t_1, t_2, \dots$  are Boolean time functions produced by the control; and  $\alpha, \beta, \dots$  are Boolean conditional coefficients with value, meaning, and constraints as shown in the table below.

$\alpha_1$	$\alpha_2$	$\beta$	$\gamma$	$\delta_1$	$\delta_2$	Condition
①	φ	φ	o	o	φ	acquisition of a new page
φ	①	o	o	φ	φ	recirculation of a page
φ	o	①	φ	φ	φ	recirculation of a page by-passing PN
o	o	φ	①	o	φ	processing of a page
o	φ	φ	o	①	φ	acquisition from storage
φ	φ	φ	φ	φ	①	storage of a page or data

The system can process in sequence all the pages through the paths  $\alpha_1$  and  $\alpha_2$ ; it can continuously process a single page, condition  $\gamma$ ; it can input and output data without involving PN through the path  $\beta$ ; it can buffer a page for a certain time in the auxiliary page array  $\Omega'_N$  through the transfers  $\delta_2$ ; it can produce a new page in array  $\Omega'_N$  during processing (combination of paths  $\delta_1$ ,  $\delta_2$  and  $\gamma$ ) for the execution of a subtask; it can introduce the new page into circulation through transfer  $\delta_1$ .

The registers of arrays  $\Omega_a$  and  $\Omega_p$  have one-to-one correspondence with the registers  $\Omega_N$  embedded in the programable network. The packer transfers the data in  $\Omega_p$  into the memory in an ordered form; the same order is used by the assembler to allocate the data of a page into  $\Omega_a$ . In this way each variable of a process always goes into the same register of PN, during the circulation of the page, if not otherwise prescribed by the program. The memory moves the pages as a First-Input-First-Output storage, or with a different rule if indicated by the program. These features eliminate the need for explicit addresses. Addresses and their manipulation account for a large part of the memory capacity, and for most of the overhead of conventional computers. When selective access is required by a given process, the corresponding addresses are obviously part of the variables of that process; accordingly, the packer has the further feature of using some process variables also for directing other variables to specific parts of the data structure organized in the memory.

The programable network does not have *per se* a specific operational configuration. It is a collection of registers, multifunction elements, and preferred links among them. Program words enable simultaneous links and functions in order to implement specialized structures which perform the data transformations demanded in each state of a process. A kind of microprogramming extended to its full allows the use of a large number of all possible combinations of the loose elements forming the programable network [21]. In this way, data transformations involving several variables are executed as a single large operation. Several different configurations can be implemented sequentially during one passage of a page through the network. The fact that the variables involved are all present in the network eliminates many intermediate steps and data movements that occur in conventional computers. When a process involves more variables than can be contained in PN, they are grouped in successive pages; the auxiliary register array  $\Omega'_N$ , which is also part of PN, allows the sharing or transfer of data. The coefficients  $\delta$  in expressions (1) can be applied to selected data or to the entire page.

It is interesting to note that the architecture of Figure 1 exhibits properties of many of the different architectures mentioned in section 1. The system has a pipeline configuration; while the programable network processes a page, the assembler assembles the next page, and the packer packs and routes the previous page. Parallel processing can be implemented simply by programming PN as a set of independent units, or, in virtual form, as a sequence of pages. The efficiency of a special-purpose computer can be achieved by structuring

PN according to the specific process. But, because the specialization of PN can change at each cycle, the machine is a general-purpose computer. Because of the three basic features -- the organization of jobs into independent pages, the circulation of the pages in a pipeline configuration, and the loose structure of the processor and memory -- this architecture has been named the Circulating Page Loose (CPL) system.

### 3. THE PROGRAMING LANGUAGE

The most interesting peculiarity of the architecture described in the previous section is its programability. This programability permits the execution of the processes in terms of structures devised by the user each time, rather than as simulation by means of a given structure (arithmetic unit connected to a random access memory) and instructions of a given set. Thus, here, the programming language refers to operational structures and data structures, rather than to commands and declarations.

The primitive elements that have been found sufficient to efficiently express the variety of processes we give a computer are the following:

- (i) a finite set of process variables  $x_r$ , a subset of which is indicated as  $X_q$ ;
- (ii) a finite set of input data  $u_r$ , a subset of which is indicated as  $U_q$ ;
- (iii) a finite set of output devices, and storages,  $z_r$ ; and
- (iv) a finite set of labeled process-states  $s_j$ , where a state is defined by:
  - (v) a function  $F_j$  which produces new values for a subset  $X_a$  as a function of the values in subsets  $X_b$  and  $U_c$ ,
  - (vi) a function  $T_j$  which produces the label of the next state as a function of the values of subsets  $X_d$  and  $U_e$ , and
  - (vii) a prescription  $R_j$  for routing some variables  $x_r$  to some output devices, or storages,  $z_r$ .

Time is represented as a sequence of discrete intervals  $i$ . A process is modeled as a finite-state machine represented by the following expressions, where the symbols refer to the primitives defined above:

$$\begin{aligned}
 X(i+1) &= F_{s(i)} [X(i), U(i)] \\
 s(i+1) &= T_{s(i)} [X(i+1), U(i)] \\
 s(i) &= s_1, s_2, \dots, s_j, \dots, s_k
 \end{aligned}
 \tag{1}$$

We must note that here states refer to phases of the model of the processes; they are neither the total internal states used in automata theory, nor the conditions of an implementation used in particular computers. These states are few and meaningful to the user. In each state, in general, there will be a different  $F$  and  $T$ . Functions  $F$  and  $T$  are thought of as operational networks; thus they can also be described in the form of digital words that implement those networks in a digital programable network [22]. In other words, we use the mapping

$$F \rightarrow N_F \rightarrow W_F \tag{2}$$

where F stands for a description (in any language) of a data transformation,  $N_F$  stands for an operational network performing that data transformation, and  $W_F$  stands for a digital word describing (in a language) that network. This global treatment of the data transformations gives conciseness to the modeling of processes, and the use of corresponding global words W gives conciseness to the actual programs. A finite-state machine so formulated is denoted with capital initials Finite State Machine (FSM). The FSM is the modular block of the programs.

Complex processes are modeled in the form of several concurrent FSMs, each of which may be implemented simultaneously by many pages. The routing prescriptions R allow the interaction among FSMs necessary for their concurrent work. A page transfers through the states of an FSM, and can transfer also through different FSMs. Thus, a process is modeled as an interplay of processing structures (the FSMs) with data structures (the pages). A program can be composed of a single FSM and page; or one FSM related to many pages; or several FSMs, each one related to one page; or many FSMs, each with many pages.

The user develops the FSMs in the form of state diagrams. Figure 3 shows an example. The encircled domains represent a state; the data transformation F is described inside these domains; the transition functions T are described with conditions indicated below horizontal lines and arrows pointing to the new states; the routing prescriptions R are indicated, typically, in connection with those arrows. Which notations are used for expressing the variables, the F, T, and R is irrelevant at this stage. State diagrams of this form constitute a complete description of a process. As such they also constitute complete programs for a computer that is isomorphic to the language of the FSM. When all the elements of the state diagram (both those represented by graphic means, and those described by alphanumerical symbols) are expressed in the codes of that computer, the actual object program is obtained. The object program is in the form of a set of quadruplets

$$\left[ W_1 W_F W_T W_R \right]_j \quad j = j_1, j_2, \dots, j_k \quad (3)$$

where  $W_F$  and  $W_T$  are the words that implement specialized networks performing functions F and T,  $W_R$  is a coded form of the routing prescriptions R, and  $W_1$  is a coded representation of the input data set  $U = U_c \cup U_e$ . j is the state label, and k is the total number of states involved in that program.

The state diagram is problem oriented and machine independent in the sense that any hypothetical machine can be implied in its construction. When a state diagram is expressed in the form of specific quadruplets, it becomes machine dependent. The transformation between the ideal machine (the state diagram) and the executable program (the quadruplets), that is, the mapping (2), is made by the user. Because the user is expected to be familiar with his processes, and to know the preferred choices, the resulting object programs are efficient and easily understood.

On the other hand, one may think that in this way, the user is burdened with clerical

tasks of which he is usually relieved by the compilers. But because of the isomorphism between the language of the FSM used for describing the processes and the architecture of the computer, it turns out that the user works on his problem and not on the intricacies of a computer which he is not interested in. Moreover, the results produced by the computer can be easily interpreted. As an example of the level of mechanization in which the user is involved, a program in the field of numerical solutions of partial differential equations is outlined below.

The dynamics of a hypothetical fluid are modeled in the form of an initial-value problem with boundary conditions. The analytical expressions considered are

$$\begin{aligned} \frac{\partial \eta}{\partial t} &= h_1 \frac{\partial \eta}{\partial x} + h_2 \frac{\partial \eta}{\partial y} \\ \frac{\partial \psi}{\partial t} &= h_3 \frac{\partial \psi}{\partial x} + h_4 \frac{\partial \psi}{\partial y} \\ \frac{\partial v}{\partial t} &= h_5 \frac{\partial v}{\partial x} + h_6 \frac{\partial v}{\partial y} \end{aligned} \quad (4)$$

where  $\eta, \psi$ , and  $v$  are the variables of the system, and  $h_r$  the given parameters. The chosen finite-difference approximation is given by

$$\begin{aligned} \eta_{i,j}^{n+1} &= \eta_{i,j}^n - k_1 (\eta_{i,j}^n - \eta_{i-1,j}^n) - k_2 (\eta_{i,j}^n - \eta_{i,j-1}^n) \\ \psi_{i,j}^{n+1} &= \psi_{i,j}^n - k_3 (\psi_{i,j}^n - \psi_{i-1,j}^n) - k_4 (\psi_{i,j}^n - \psi_{i,j-1}^n) \\ v_{i,j}^{n+1} &= v_{i,j}^n - k_5 (v_{i,j}^n - v_{i-1,j}^n) - k_6 (v_{i,j}^n - v_{i,j-1}^n) \end{aligned} \quad (5)$$

with the conventions:

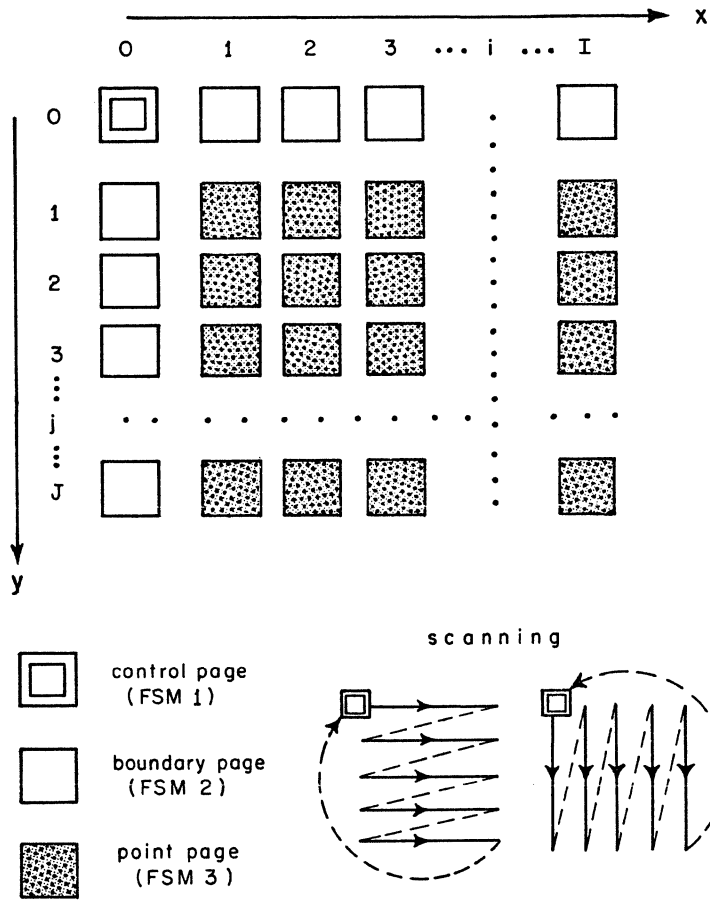
$$\begin{aligned} x &= i \Delta x & i &= 1, 2, \dots, I \\ y &= j \Delta y & j &= 1, 2, \dots, J \\ t &= n \Delta t & n &= 1, 2, \dots, N \end{aligned}$$

and the  $k_r$  derived from the  $h_r$ .

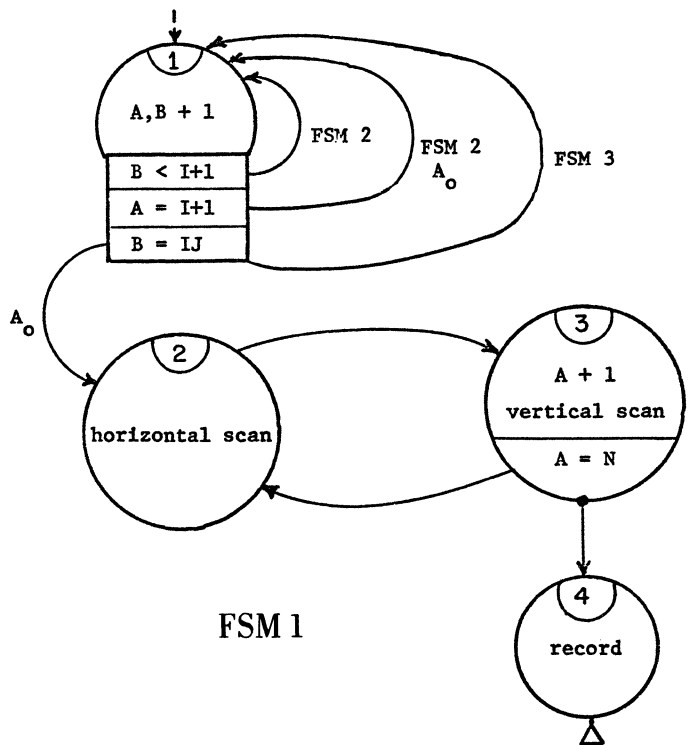
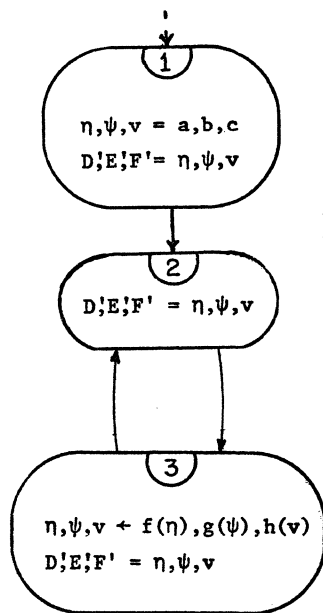
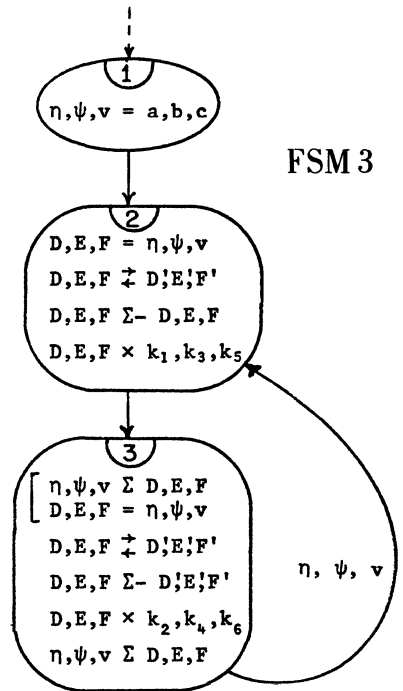
To obtain the solution, an abstract machine is conceived, that has a page for each point of the two dimensional space of the system (Figure 2), a page for each boundary point, and a control page. The symbols  $\eta, \psi$ , and  $v$ , related to the variables of the process, are considered as names for three variables  $x_r$ ; initial values a, b, and c of these variables are treated as input data  $u_r$ ; the parameters  $k_r$  also are treated as input data. Moreover, an additional three variables  $x_r$ , named D, E and F, are used for temporary purposes. The pages related to the points of the fluid perform an FSM 3 as described in Figure 3, which implements expressions (5); the pages related to the boundary points perform an FSM 2 which implements a time evolution of the boundary values; and the control page performs an FSM 1 which controls the work of the entire system. The pages circulate in the structure shown in Figure 1, with different scanning as indicated in Figure 2.

Even without entering into the details of the language of the programable network, Figure 3 should convey the level of abstraction of these operational structures. For instance, FSM 1, which constructs and controls the entire

**FIG 2**



**FIG 3**



machine, has four states. State 1 is devoted to creating the page array indicated in Figure 2. In this state, function F consists simply in incrementing variables A and B by one. Function T is expressed as a self-explanatory decision table (the transition from the corner corresponds to the "else" condition). The routing is different for the different transitions and consists of creating pages related to given FSMs and in clearing variable A. State 2 prescribes a horizontal scanning of the pages. State 3 prescribes a vertical scanning, and provides for the test of the number of time steps. State 4 orders an output record of the computed quantities, and makes the pages disappear (transition to a triangle).

The computation of the variables  $\eta$ ,  $\psi$ , and  $v$  at each point (FSM 3) is obtained by means of simple networks of a parallel nature established by the user in accordance with expressions (5). As an example, in state 1 of FSM 3, the input set U, consisting of the data a, b, and c, is transferred in parallel into the set X, consisting of the variables  $\eta$ ,  $\psi$ , and  $v$ . In state 3 of FSM 3, there is a succession of five networks: the first produces simultaneously the accumulation of the original values of D, E, F into  $\eta$ ,  $\psi$ ,  $v$ , and the transfer of the original values of  $\eta$ ,  $\psi$ ,  $v$  into D, E, F; the second produces an interchange of values between D, E, F and D', E', F', which are variables that remain in  $\Omega'_N$  of the network during the circulation of the pages; the third produces the subtraction of D', E', F' from D, E, F; the fourth produces the multiplication of D, E, F by the data set  $k_2$ ,  $k_4$ ,  $k_6$ ; and the fifth the accumulation of the present values of D, E, F into  $\eta$ ,  $\psi$ ,  $v$ . A routing prescription sends the present values of  $\eta$ ,  $\psi$ ,  $v$  to an output storage.

Obviously, the interest for such constructs is not to make the user do what can be provided by a compiler, but to give the user the possibility either of providing what has not been anticipated by the software systems, or of obtaining specific optimizations. In this example, the aim was to minimize the memory and the execution time. The entire computation is made with  $6IJ + 3(I+J) + 2$  memory words. The machine cycles are  $(2N+2)(I+1)(J+1)$ , with an average of four to five networks per cycle.

#### 4. RESULTS AND DISCUSSION

The results obtained from the use of this architecture for processing in real-time radar signals have already been reported [23,24,25,26]. Obviously, in these applications, advantage is derived from the capability of the programable network to perform complex operations in one cycle, and to structure the memory in accordance to the stream of data. An application of significant interest is a program for processing weather-radar signals in real time that discriminates weather echoes from ground echoes, during the normal operation of the radar.

The easy interaction between user and computer is also very significant. The fact that the same FSM is both the model of the process used by the user and the program actually executed by the computer, makes it possible to develop a program in "real time" as suggested by the results. In the line of the mechanization of Figure 3, programs have been experimented that acquire actual initial data, in real

time, from a weather radar, and then produce different evolutions of the precipitation pattern in terms of the values of parameters set by the user at each time, or modifications of the programs.

In research work, data processing is typically achieved today by means of systems comprising several special-purpose units and a general-purpose computer. The former efficiently execute the particular data transformations demanded in the process, and the latter provides for the computation and the control of the entire system. In these cases, a single computer with a CPL architecture could advantageously perform all these activities. The programable network is capable of executing both the particular data transformations and the computations; the programs in the FSM form are particularly suitable for controlling complex activities; and the organization of data in the form of pages makes efficient use of the memory capacity.

Another field for which this architecture is particularly efficient is that in which differential analyzers were advantageously used [27]. The PN can be programed in the form of integrators, and each page takes on the role of a term in a system of equations. Transformations such as the Fast Fourier Transform, similarly, can be executed efficiently by properly organizing the pages and configuring PN for complex butterflies [28]. This architecture has also been suggested for the computers in an integrated telecommunication network [29].

But the fact that this architecture accepts directly programs expressed in the FSM form and these programs correspond to the image of the process as developed by the user, triggers a more general interest in this approach [30]. The next subject of study that seems deserving of attention is the feasibility of a programming language that shares the flexibility and conciseness of the FSM and the adaptability to different forms of expression offered by the well-established use of compilers.

As far as the hardware software trade-off is concerned, this architecture constitutes an interesting new approach. The programability of the hardware configuration allows the efficiency peculiar to the hardware implementations together with the flexibility characteristic of the software implementations. Moreover, the description of these configurations is also interesting as a programming language *per se*. In using the first machine constructed with this architecture [31], we consistently find that programs in the form of FSM are much simpler than the equivalent programs in conventional machine language, and they have a complexity comparable to that of programs expressed in high level language. For complex processes, the programs in FSM form seem to be simpler than the equivalent ones in high level language. This finding has an interesting similarity with von Neumann's contention that for complex automata the description of an automaton is simpler than the description of the process performed by the automaton [32].

#### ACKNOWLEDGEMENT

This work has been supported, at different times, by the National Aeronautics and Space



Administration under contracts NASW-2276,  
NSR-09-015-033, and NASr-158.

#### REFERENCES

1. Slotnick, D.L., W.C. Borck, and R.C. McReynolds, "The Solomon Computer", Proc. Fall Joint Computer Conf., 97-107, 1962.
2. Holland, J.H., "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously", Proc. Eastern Joint Computer Conf., 108-113, 1959.
3. Unger, S.H., "A Computer Oriented Toward Spatial Problems", Proc. IRE, 1744-1750, 1958.
4. Estrin, G., "Organization of Computer Systems - The Fixed Plus Variable Structure Computer", Proc. Western Joint Computer Conf., 33-40, 1960.
5. Hobbs, L.C., and al. (Ed.), Parallel Processor Systems, Technologies, and Applications, Spartan Books, N.Y., 1970.
6. Chen, T.C., "Parallelism, Pipelining and Computer Efficiency", Comp. Res., 10, 69-74, 1971.
7. Ramarmoorthy, C.V., and S.S. Reddi, "Towards a Theory of Pipelined Computing Systems", Proc. 10th Allerton Conf. on Circuit and System Theory, University of Illinois, Urbana, Ill., 759-768, 1972.
8. Buchholz, W. (Ed.), Planning a Computer System, McGraw-Hill, N.J., 1962.
9. Hallin, T.G., and M.J. Flynn, "Pipelining of Arithmetic Functions", IEEE Trans. C-21, 880-886, 1972.
10. Cotten, L.W., "Maximum-Rate Pipeline Systems", Proc. Spring Joint Computer Conf., 581-586, 1969.
11. Graham, W.R., "The Parallel and the Pipeline Computers", Datamation, Vol. 16, April, 1970.
12. Flynn, M.J., "Some Computer Organizations and their Effectiveness", IEEE Trans. C-21, 948-960, 1972.
13. McKeeman, W.M., "Language Directed Computer Design", Proc. FJCC, AFIPS Vol. 31, 413-417, 1967.
14. Anderson, J.P., "A Computer for Direct Execution of Algorithmic Languages", Proc. Eastern JCC, 184-193, 1961.
15. Bashkow, T.R., A. Sasson, and A. Kronfeld, "System Design of a Fortran Machine", IEEE Trans. EC-16, 485-499, 1967.
16. Weber, H., "A Microprogrammed Implementation of EULER on IBM System/360 Model 30", Comm. ACM, 10, 549-558, 1967.
17. Thurber, K.J., and J.M. Myrna, "System Design for a Cellular APL Computer", IEEE Trans. C-19, 291-300, 1970.
18. Chelsey, G.D., and W.A. Smith, "The Hardware-Implemented High-Level Machine Language for SYMBOL", Proc. SJCC, AFIPS Vol. 39, 563-573, 1971.
19. Hassitt, A., J.W. Lageschulte, and L.E. Lyon, "Implementation of a High Level Language Machine", Comm. ACM, 16, 199-212, 1973.
20. Barte, T.C., I.L. Lebow, and I.S. Reed, Theory and Design of Digital Machines, McGraw-Hill Co., New York, 1962.
21. Schaffner, M.R., "A System with Programmable Hardware", Digest 5th IEEE Int. Computer Conf., Boston Mass., 17-18, 1971.
22. Schaffner, M.R., "A Procedure for Describing Discrete Processes", Proc. 10th Allerton Conf. on Circuit and System Theory, Urbana, Ill., 462-470, 1972.
23. Austin, P.M., and M.R. Schaffner, "Computations and Experiments Relevant to Digital Processing Weather-Radar Echoes", Prepr. 14th Weather Radar Conf., 375-380, 1970.
24. Schaffner, M.R., "Computers Formed by the Problems, rather than Problems Deformed by the Computers", Digest 6th IEEE Int. Computer Conf., San Francisco, Cal., 259-264, 1972.
25. Schaffner, M.R., "On the Data Processing for Weather Radar", Prepr. 15th Conf. on Radar Meteor., 368-373, 1972.
26. Schaffner, M.R., "Echo Movement and Evolution from Real-Time Processing", Prepr. 15th Radar Meteor. Conf., 374-348, 1972.
27. Sizer, T.R.H., (Ed.), The Digital Differential Analyzer, Chapman and Hall, Ltd., London, 1968.
28. Schaffner, M.R., "Study of the Applicability of the CPL System to Doppler Radar Signal Processing", National Center for Atmospheric Research, Boulder, Col., 1973.
29. Cappetti, I., and M.R. Schaffner, "Structure of a Communication Network and Its Control Computers", Proc. Symp. Computer-Communications Networks and Teletraffic, M.R.I., Vol. XXII, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 1972.
30. Schaffner, M.R., "Study of a Self-Organizing Computer", Final Report, Contract NASW-2276, National Aeronautics Space Administration, 1973.
31. Schaffner, M.R., "A Computer Modeled After an Automaton", Proc. Symp. Computers and Automata, M.R.I. Symp., Vol. XXI, Polytechnic Institute of Brooklyn, Brooklyn, N.Y., 635-650, 1971.
32. von Neumann, J., "The General and Logical Theory of Automata", Hixon Symposium, 1948, Pasadena, Cal.; reprinted in John von Neumann, Collected Works, (A.H. Taub, Ed.), Pergamon Press, 1963.

